

Vue.js

目錄

vue.js 中文文档 1.0

来源：Vue.js

教程

起步

我们以 Vue 数据绑定的快速导览开始。如果你对高级概述更感兴趣，可查看这篇博文。

尝试 Vue.js 最简单的方法是使用 JSFiddle Hello World 例子。在浏览器新标签页中打开它，跟着我们查看一些基础示例。如果你喜欢用包管理器下载/安装，查看安装教程。

Hello World

```
<div id="app">
 {{ message }}
</div>
```

```
new Vue({
 el: '#app',
 data: {
 message: 'Hello Vue.js!'
 }
})
```

Hello Vue.js!

双向绑定

```
<div id="app">
 <p>{{ message }}</p>
 <input v-model="message">
</div>
```

```
new Vue({
 el: '#app',
 data: {
 message: 'Hello Vue.js!'
 }
})
```

Hello Vue.js!

Hello Vue.js!

渲染列表

```
<div id="app">
 <ul>
 <li v-for="todo in todos">
 {{ todo.text }}
 </li>
 </ul>
</div>
```

```
new Vue({
 el: '#app',
 data: {
 todos: [
 { text: 'Learn JavaScript' },
 { text: 'Learn Vue.js' },
 { text: 'Build Something Awesome' }
 ]
 }
})
```

- Learn JavaScript
- Learn Vue.js
- Build Something Awesome

处理用户输入

```
<div id="app">
 <p>{{ message }}</p>
 <button v-on:click="reverseMessage">Reverse Message</button>
</div>
```

```
new Vue({
 el: '#app',
 data: {
 message: 'Hello Vue.js!'
 },
 methods: {
 reverseMessage: function () {
 this.message = this.message.split('').reverse().join('')
 }
 }
})
```

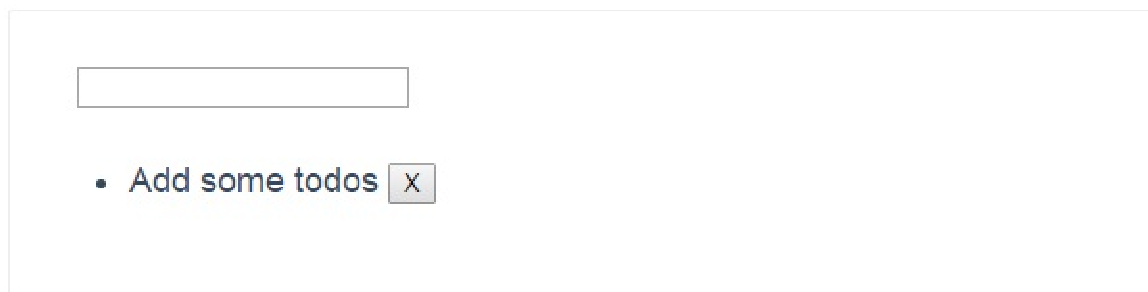
Hello Vue.js!

Reverse Message

综合

```
<div id="app">
 <input v-model="newTodo" v-on:keyup.enter="addTodo">
 <ul>
 <li v-for="todo in todos">
 <span>{{ todo.text }}</span>
 <button v-on:click="removeTodo($index)">X</button>
 </li>
 </ul>
</div>
```

```
new Vue({
 el: '#app',
 data: {
 newTodo: '',
 todos: [
 { text: 'Add some todos' }
 ]
 },
 methods: {
 addTodo: function () {
 var text = this.newTodo.trim()
 if (text) {
 this.todos.push({ text: text })
 this.newTodo = ''
 }
 },
 removeTodo: function (index) {
 this.todos.splice(index, 1)
 }
 }
})
```



希望上例能让你对 Vue.js 的工作原理有一个基础概念。我知道你现在有许多疑问——继续阅读，在后面的教程将一一解答。

概述

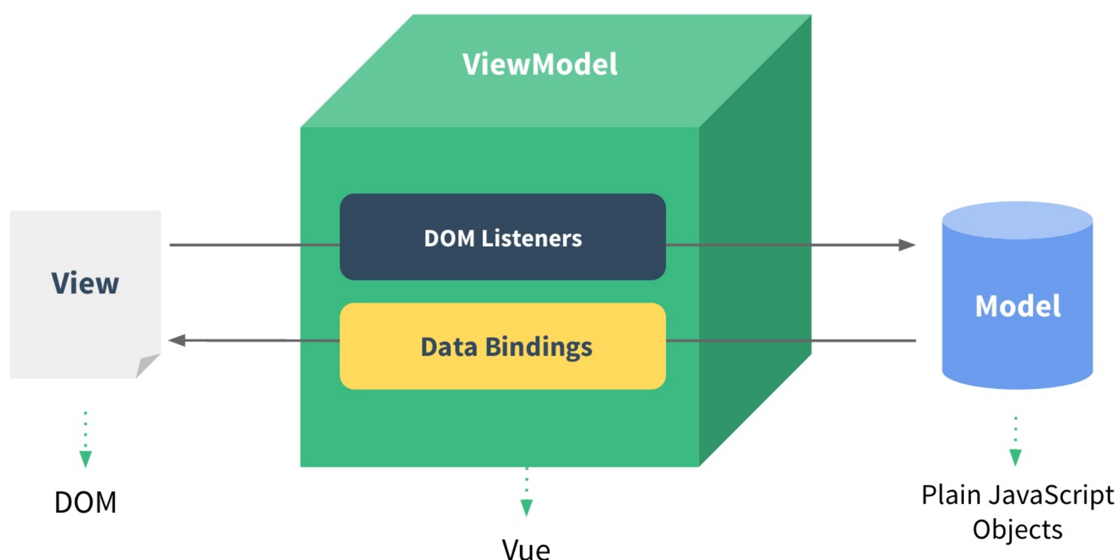
Vue.js（读音 /vju?/, 类似于 **view**）是一个构建数据驱动的 web 界面的库。Vue.js 的目标是通过尽可能简单的 API 实现响应的数据绑定和组合的视图组件。

Vue.js 自身不是一个全能框架——它只聚焦于视图层。因此它非常容易学习，非常容易与其它库或已有项目整合。另一方面，在与相关工具和支持库一起使用时，Vue.js 也能完美地驱动复杂的单页应用。

如果你是有经验的前端开发者，想知道 Vue.js 与其它库/框架的区别，查看对比其它框架；如果你对使用 Vue.js 开发大型应用更感兴趣，查看构建大型应用。

响应的数据绑定

Vue.js 的核心是一个响应的数据绑定系统，它让数据与 DOM 保持同步非常简单。在使用 jQuery 手工操作 DOM 时，我们的代码常常是命令式的、重复的与易错的。Vue.js 拥抱数据驱动的视图概念。通俗地讲，它意味着我们在普通 HTML 模板中使用特殊的语法将 DOM “绑定”到底层数据。一旦创建了绑定，DOM 将与数据保持同步。每当修改了数据，DOM 便相应地更新。这样我们应用中的逻辑就几乎都是直接修改数据了，不必与 DOM 更新搅在一起。这让我们的代码更容易撰写、理解与维护。



可能是最简单的例子：

```
<!-- 这是我们的 View -->
<div id="example-1">
 Hello {{ name }}!
</div>
```

```
// 这是我们的 Model
var exampleData = {
 name: 'Vue.js'
}

// 创建一个 Vue 实例或 "ViewModel"
// 它连接 View 与 Model
var exampleVM = new Vue({
 el: '#example-1',
 data: exampleData
})
```

结果：

Hello Vue.js!

看起来这跟单单渲染一个模板非常类似，但是 Vue.js 在背后做了大量工作。并且 DOM 会自动响应数据的变化。我们如何知道？打开你的浏览器的控制台，修改 `exampleData.name` ，你将看到上例相应地更新。

注意我们不需要撰写任何 DOM 操作代码：被绑定增强的 HTML 模板是底层数据状态的声明式的映射，数据不过是普通 JavaScript 对象。我们的视图完全由数据驱动。

让我们来看第二个例子：

```
<div id="example-2">
 <p v-if="greeting">Hello!</p>
</div>
```

```
var exampleVM2 = new Vue({
 el: '#example-2',
 data: {
 greeting: true
 }
})
```

Hello!

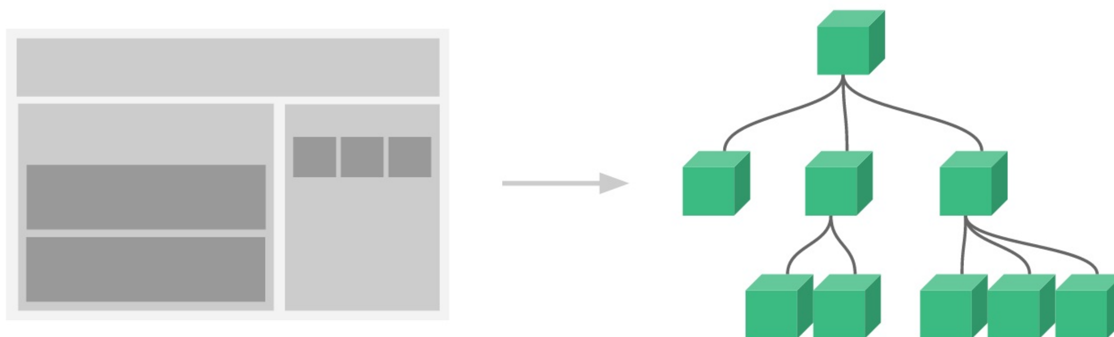
这里我们遇到新东西。你看到的 `v-if` 特性被称为指令。指令带有前缀 `v-` ，以指示它们是 Vue.js 提供的特殊特性。并且如你所想象的，它们会对绑定的目标元素添加响应式的特殊行为。继续在控制台设置 `exampleVM2.greeting` 为 `false` ，你会发现 “Hello!” 消失了。

第二个例子演示了我们不仅可以绑定 DOM 文本到数据，也可以绑定 DOM 结构 到数据。而且，Vue.js 也提供一个强大的过渡效果系统，可以在 Vue 插入/删除元素时自动应用过渡效果。

也有一些其它指令，每个都有特殊的功能。例如 `v-for` 指令用于显示数组元素， `v-bind` 指令用于绑定 HTML 特性。我们将在后面详细讨论全部的数据绑定语法。

组件系统

组件系统是 Vue.js 另一个重要概念，因为它提供了一种抽象，让我们可以用独立可复用的小组件来构建大型应用。如果我们考虑到这点，几乎任意类型的应用的界面都可以抽象为一个组件树：



实际上，一个典型的用 Vue.js 构建的大型应用将形成一个组件树。在后面的教程中我们将详述组件，不过这里有一个假想的例子，看看使用了组件的应用模板是什么样的：

```
<div id="app">
 <app-nav></app-nav>
 <app-view>
 <app-sidebar></app-sidebar>
 <app-content></app-content>
 </app-view>
</div>
```

你可能已经注意到 Vue.js 组件非常类似于自定义元素——它是 Web 组件规范的一部分。实际上 Vue.js 的组件语法参考了该规范。例如 Vue 组件实现了 Slot API 与 `is` 特性。但是，有几个关键的不同：

1. Web 组件规范仍然远未完成，并且没有浏览器实现。相比之下，Vue.js 组件不需要任何补丁，并且在所有支持的浏览器（IE9 及更高版本）之下表现一致。必要时，Vue.js 组件也可以放在原生自定义元素之内。

2. Vue.js 组件提供了原生自定义元素所不具备的一些重要功能，比如组件间的数据流，自定义事件系统，以及动态的、带特效的组件替换。

组件系统是用 Vue.js 构建大型应用的基础。另外，Vue.js 生态系统也提供了高级工具与多种支持库，它们和 Vue.js 一起构成了一个更加“框架”性的系统。

Vue 实例

构造器

每个 Vue.js 应用的起步都是通过构造函数 `Vue` 创建一个 **Vue** 的根实例：

```
var vm = new Vue({
  // 选项
})
```

一个 Vue 实例其实正是一个 MVVM 模式中所描述的 ViewModel - 因此在文档中经常会使用 `vm` 这个变量名。

在实例化 Vue 时，需要传入一个选项对象，它可以包含数据、模板、挂载元素、方法、生命周期钩子等选项。全部的选项可以在 API 文档中查看。

可以扩展 `Vue` 构造器，从而用预定义选项创建可复用的组件构造器：

```
var MyComponent = Vue.extend({
  // 扩展选项
})

// 所有的 `MyComponent` 实例都将以预定义的扩展选项被创建
var myComponentInstance = new MyComponent()
```

尽管可以命令式地创建扩展实例，不过在多数情况下将组件构造器注册为一个自定义元素，然后声明式地用在模板中。我们将在后面详细说明组件系统。现在你只需知道所有的 Vue.js 组件其实都是被扩展的 Vue 实例。

属性与方法

每个 Vue 实例都会代理其 `data` 对象里所有的属性：

```
var data = { a: 1 }
var vm = new Vue({
 data: data
})

vm.a === data.a // -&gt; true

// 设置属性也会影响到原始数据
vm.a = 2
data.a // -&gt; 2

// ... 反之亦然
data.a = 3
vm.a // -&gt; 3
```

注意只有这些被代理的属性是响应的。如果在实例创建之后添加新的属性到实例上，它不会触发视图更新。我们将在后面详细讨论响应系统。

除了这些数据属性，Vue 实例暴露了一些有用的实例属性与方法。这些属性与方法都有前缀 `$`，以便与代理的数据属性区分。例如：

```
var data = { a: 1 }
var vm = new Vue({
 el: '#example',
 data: data
})

vm.$data === data // -&gt; true
vm.$el === document.getElementById('example') // -&gt; true

// $watch 是一个实例方法
vm.$watch('a', function (newVal, oldVal) {
 // 这个回调将在 `vm.a`  改变后调用
})
```

参考 API 文档查看全部的实例属性与方法。

实例生命周期

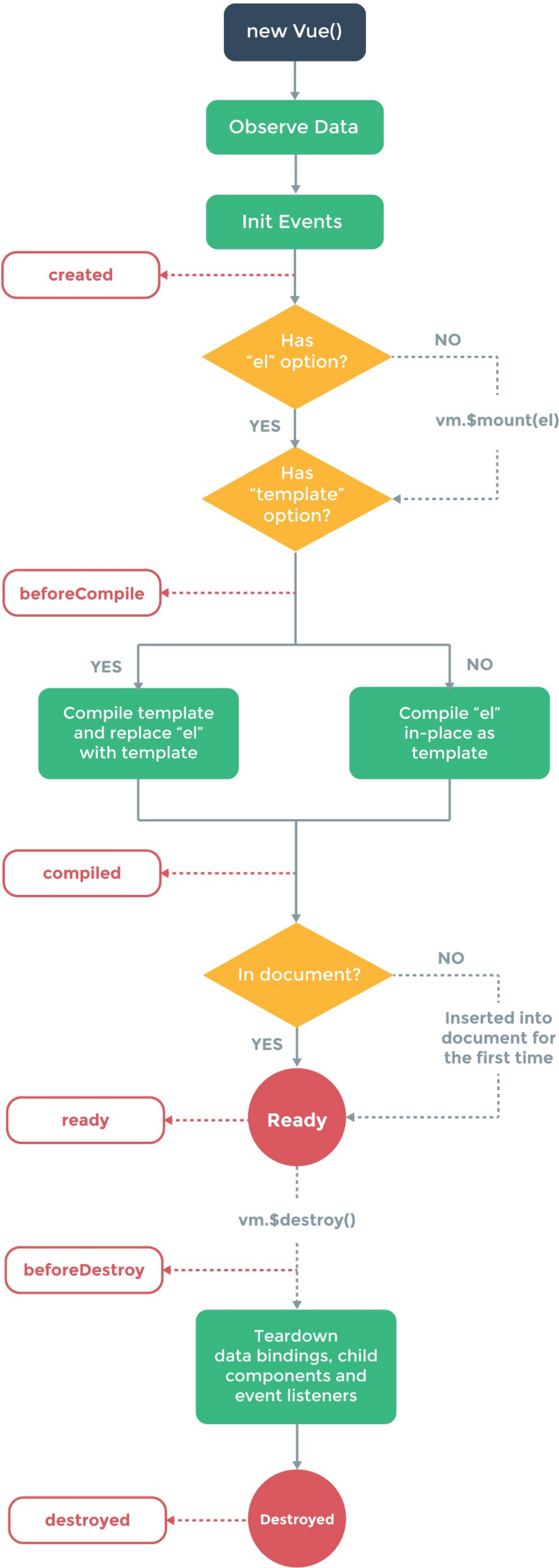
Vue 实例在创建时有一系列初始化步骤——例如，它需要建立数据观察，编译模板，创建必要的数据绑定。在此过程中，它也将调用一些生命周期钩子，给自定义逻辑提供运行机会。例如 `created` 钩子在实例创建后调用：

```
var vm = new Vue({
 data: {
 a: 1
 },
 created: function () {
 // `this` 指向 vm 实例
 console.log('a is: ' + this.a)
 }
})
// -&gt; "a is: 1"
```

也有一些其它的钩子，在实例生命周期的不同阶段调用，如 `compiled` 、 `ready` 、 `destroyed` 。钩子的 `this` 指向调用它的 Vue 实例。一些用户可能会问 Vue.js 是否有“控制器”的概念？答案是，没有。组件的自定义逻辑可以分割在这些钩子中。

生命周期图示

下图说明了实例的生命周期。你不需要立马弄明白所有的东西，不过以后它会有帮助。


new Vue()
Observe Data
Init Events
created
Has
"el" option?
NO
YES
vm.$mount(el)
Has
"template"
option?
beforeCompile
YES
NO
Compile template
and replace "el"
with template
Compile "el"
in-place as
template
compiled
In document?
NO
YES
Inserted into
document for
the first time
ready
Ready
vm.$destroy()
beforeDestroy
Teardown
data bindings, child
components and
event listeners
destroyed
Destroyed

数据绑定语法

Vue.js 的模板是基于 DOM 实现的。这意味着所有的 Vue.js 模板都是可解析的有效的 HTML，且通过一些特殊的特性做了增强。Vue 模板因而从根本上不同于基于字符串的模板，请记住这点。

插值

文本

数据绑定最基础的形式是文本插值，使用 “Mustache” 语法（双大括号）：

```
<span>Message: {{ msg }}</span>
```

Mustache 标签会被相应数据对象的 `msg` 属性的值替换。每当这个属性变化时它也会更新。

你也可以只处理单次插值，今后的数据变化就不会再引起插值更新了：

```
<span>This will never change: {{* msg }}</span>
```

原始的 HTML

双 Mustache 标签将数据解析为纯文本而不是 HTML。为了输出真的 HTML 字符串，需要用三 Mustache 标签：

```
<div>{{{ raw_html }}}</div>
```

内容以 HTML 字符串插入——数据绑定将被忽略。如果需要复用模板片断，应当使用 partials。

在网站上动态渲染任意 HTML 是非常危险的，因为容易导致 XSS 攻击。记住，只对可信内容使用 HTML 插值，永不用于用户提交的内容。

HTML 特性

Mustache 标签也可以用在 HTML 特性 (Attributes) 内：

```
<div id="item-{{ id }}"></div>
```

注意在 Vue.js 指令和特殊特性内不能用插值。不必担心，如果 Mustache 标签用错了地方 Vue.js 会给出警告。

绑定表达式

放在 Mustache 标签内的文本称为绑定表达式。在 Vue.js 中，一段绑定表达式由一个简单的 JavaScript 表达式和可选的一个或多个过滤器构成。

JavaScript 表达式

到目前为止，我们的模板只绑定到简单的属性键。不过实际上 Vue.js 在数据绑定内支持全功能的 JavaScript 表达式：

```
{{ number + 1 }}

{{ ok ? 'YES' : 'NO' }}

{{ message.split('').reverse().join('') }}
```

这些表达式将在所属的 Vue 实例的作用域内计算。一个限制是每个绑定只能包含单个表达式，因此下面的语句是无效的：

```
<!-- 这是一个语句，不是一个表达式： -->
{{ var a = 1 }}

<!-- 流程控制也不可以，可改用三元表达式 -->
{{ if (ok) { return message } }}
```

过滤器

Vue.js 允许在表达式后添加可选的“过滤器 (Filter) ”，以“管道符”指示：

```
{{ message &#124; capitalize }}
```

这里我们将表达式 `message` 的值“管输（pipe）”到内置的 `capitalize` 过滤器，这个过滤器其实只是一个 JavaScript 函数，返回大写化的值。Vue.js 提供数个内置过滤器，在后面我们会谈到如何开发自己的过滤器。

注意管道语法不是 JavaScript 语法，因此不能在表达式内使用过滤器，只能添加到表达式的后面。

过滤器可以串联：

```
{{ message &#124; filterA &#124; filterB }}
```

过滤器也可以接受参数：

```
{{ message &#124; filterA 'arg1' arg2 }}
```

过滤器函数始终以表达式的值作为第一个参数。带引号的参数视为字符串，而不带引号的参数按表达式计算。这里，字符串 `'arg1'` 将传给过滤器作为第二个参数，表达式 `arg2` 的值在计算出来之后作为第三个参数。

指令

指令 (Directives) 是特殊的带有前缀 `v-` 的特性。指令的值限定为绑定表达式，因此上面提到的 JavaScript 表达式及过滤器规则在这里也适用。指令的职责就是当其表达式的值改变时把某些特殊的行为应用到 DOM 上。我们来回头看下“概述”里的例子：

```
<p v-if="greeting">Hello!</p>
```

这里 `v-if` 指令将根据表达式 `greeting` 值的真假删除/插入 `<p>` 元素。

参数

有些指令可以在其名称后面带一个“参数” (Argument)，中间放一个冒号隔开。例如， `v-bind` 指令用于响应地更新 HTML 特性：

```
<a v-bind:href="url"></a>
```

这里 `href` 是参数，它告诉 `v-bind` 指令将元素的 `href` 特性跟表达式 `url` 的值绑定。可能你已注意到可以用特性插值 `href="{{url}}"` 获得同样的结果：这样没错，并且实际上在内部特性插值会转为 `v-bind` 绑定。

另一个例子是 `v-on` 指令，它用于监听 DOM 事件：

```
<a v-on:click="doSomething">
```

Here the argument is the event name to listen to. We will talk about event handling in more details too. 这里参数是被监听的事件的名字。我们也会详细说明事件绑定。

修饰符

修饰符 (Modifiers) 是以半角句号 `.` 开始的特殊后缀，用于表示指令应当以特殊方式绑定。例如 `.literal` 修饰符告诉指令将它的值解析为一个字面字符串而不是一个表达式：

```
<a v-bind:href.literal="/a/b/c"></a>
```

当然，这似乎没有意义，因为我们只需要使用 `href="/a/b/c"` 而不必使用一个指令。这个例子只是为了演示语法。后面我们将看到修饰符更多的实践用法。

缩写

`v-` 前缀是一种标识模板中特定的 Vue 特性的视觉暗示。当你需要在一些现有的 HTML 代码中添加动态行为时，这些前缀可以起到很好的区分效果。但你在使用一些常用指令的时候，你会感觉一直这么写实在是啰嗦。而且在构建单页应用时，Vue.js 会管理所有的模板，此时 `v-` 前缀也没那么重要了。因此Vue.js 为两个最常用的指令 `v-bind` 和 `v-on` 提供特别的缩写：

`v-bind` 缩写

```
<!-- 完整语法 -->
<a v-bind:href="url"></a>

<!-- 缩写 -->
<a :href="url"></a>

<!-- 完整语法 -->
<button v-bind:disabled="someDynamicCondition">Button</button>

<!-- 缩写 -->
<button :disabled="someDynamicCondition">Button</button>
```

`v-on` 缩写

```
<!-- 完整语法 -->
<a v-on:click="doSomething"></a>

<!-- 缩写 -->
<a @click="doSomething"></a>
```

它们看起来跟“合法”的 HTML 有点不同，但是它们在所有 Vue.js 支持的浏览器中都能被正确地解析，并且不会出现在最终渲染的标记中。缩写语法完全是可选的，不过随着一步步学习的深入，你会庆幸拥有它们。

计算属性

在模板中表达式非常便利，但是它们实际上只用于简单的操作。模板是为了描述视图的结构。在模板中放入太多的逻辑会让模板过重且难以维护。这就是为什么 Vue.js 将绑定表达式限制为一个表达式。如果需要多于一个表达式的逻辑，应当使用计算属性。

基础例子

```
&lt;div id="example"&gt;
 a={{ a }}, b={{ b }}
&lt;/div&gt;
```

```
var vm = new Vue({
 el: '#example',
 data: {
 a: 1
 },
 computed: {
 // 一个计算属性的 getter
 b: function () {
 // `this` 指向 vm 实例
 return this.a + 1
 }
 }
})
```

结果：

a=1, b=2

这里我们声明了一个计算属性 `b` 。我们提供的函数将用作属性 `vm.b` 的 getter。

```
console.log(vm.b) // -&gt; 2
vm.a = 2
console.log(vm.b) // -&gt; 3
```

你可以打开浏览器的控制台，修改例子的 vm。 `vm.b` 的值始终取决于 `vm.a` 的值。

你可以像绑定普通属性一样在模板中绑定计算属性。Vue 知道 `vm.b` 依赖于 `vm.a` ，因此当 `vm.a` 发生改变时，依赖于 `vm.b` 的绑定也会更新。而且最妙的是我们是声明式地创建这种依赖关系：计算属性的 getter 是干净无副作用的，因此也是易于测试和理解的。

计算属性 vs. $watch

Vue.js 提供了一个方法 `$watch` ，它用于观察 Vue 实例上的数据变动。当一些数据需要根据其它数据变化时， `$watch` 很诱人 —— 特别是如果你来自 AngularJS。不过，通常更好的办法是使用计算属性而不是一个命令式的 `$watch` 回调。考虑下面例子：

```
&lt;div id="demo"&gt;{{fullName}}&lt;/div&gt;
```

```
var vm = new Vue({
 data: {
 firstName: 'Foo',
 lastName: 'Bar',
 fullName: 'Foo Bar'
 }
})

vm.$watch('firstName', function (val) {
 this.fullName = val + ' ' + this.lastName
})

vm.$watch('lastName', function (val) {
 this.fullName = this.firstName + ' ' + val
})
```

上面代码是命令式的重复的。跟计算属性对比：

```
var vm = new Vue({
 data: {
 firstName: 'Foo',
 lastName: 'Bar'
 },
 computed: {
 fullName: function () {
 return this.firstName + ' ' + this.lastName
 }
 }
})
```

更好，不是吗？

计算 setter

计算属性默认只是 getter，不过在需要时你也可以提供一个 setter：

```
// ...
computed: {
 fullName: {
 // getter
 get: function () {
 return this.firstName + ' ' + this.lastName
 },
 // setter
 set: function (newValue) {
 var names = newValue.split(' ')
 this.firstName = names[0]
 this.lastName = names[names.length - 1]
 }
 }
}
// ...
```

现在在调用 `vm.fullName = 'John Doe'` 时，setter 会被调用， `vm.firstName` 和 `vm.lastName` 也会有相应更新。

关于计算属性背后的原理和技术细节详见响应系统介绍中的具体章节。

Class 与 Style 绑定

数据绑定一个常见需求是操作元素的 class 列表和它的内联样式。因为它们都是属性，我们可以用 `v-bind` 处理它们：我们只需要计算出表达式最终的字符串。不过，字符串拼接麻烦又易错。因此，在 `v-bind` 用于 `class` 和 `style` 时，Vue.js 专门增强了它。表达式的结果类型除了字符串之外，还可以是对象或数组。

绑定 HTML Class

尽管可以用 Mustache 标签绑定 class，比如 `class="{{ className }}"`，但是我们不推荐这种写法和 `v-bind:class` 混用。两者只能选其一！

对象语法

我们可以传给 `v-bind:class` 一个对象，以动态地切换 class。注意 `v-bind:class` 指令可以与普通的 `class` 特性共存：

```
<div class="static" v-bind:class="{ 'class-a': isA, 'class-b': isB
```

```
data: {
 isA: true,
 isB: false
}
```

渲染为：

```
<div class="static class-a"></div>
```

当 `isA` 和 `isB` 变化时，class 列表将相应地更新。例如，如果 `isB` 变为 `true`，class 列表将变为 `"static class-a class-b"`。

你也可以直接绑定数据里的一个对象：

```
<div v-bind:class="classObject"></div>
```

```
data: {
 classObject: {
 'class-a': true,
 'class-b': false
 }
}
```

我们也可以在这里绑定一个返回对象的计算属性。这是一个常用且强大的模式。

数组语法

我们可以把一个数组传给 `v-bind:class` ，以应用一个 class 列表：

```
<div v-bind:class="[classA, classB]">
```

```
data: {
 classA: 'class-a',
 classB: 'class-b'
}
```

渲染为：

```
<div class="class-a class-b"></div>
```

如果你也想根据条件切换列表中的 class，可以用三元表达式：

```
<div v-bind:class="[classA, isB ? classB : '']">
```

此例始终添加 `classA` ，但是只有在 `isB` 是 `true` 时添加 `classB` 。

绑定内联样式

对象语法

`v-bind:style` 的对象语法十分直观——看着非常像 CSS，其实它是一个 JavaScript 对象。CSS 属性名可以用驼峰式（camelCase）或短横分隔命名（kebab-case）：

```
<div v-bind:style="{ color: activeColor, fontSize: fontSize + 'px'
```

```
data: {
 activeColor: 'red',
 fontSize: 30
}
```

直接绑定到一个样式对象通常更好，让模板更清晰：

```
<div v-bind:style="styleObject"></div>
```

```
data: {
 styleObject: {
 color: 'red',
 fontSize: '13px'
 }
}
```

同样的，对象语法常常结合返回对象的计算属性使用。

数组语法

`v-bind:style` 的数组语法可以将多个样式对象应用到一个元素上：

```
<div v-bind:style="[styleObjectA, styleObjectB]">
```

自动添加前缀

当 `v-bind:style` 使用需要厂商前缀的 CSS 属性时，如 `transform` ，Vue.js 会自动侦测并添加相应的前缀。

条件渲染

v-if

在字符串模板中，如 Handlebars，我们得像这样写一个条件块：

```
<!-- Handlebars 模板 -->
{{#if ok}}
 <h1>Yes</h1>
{{/if}}
```

在 Vue.js，我们使用 `v-if` 指令实现同样的功能：

```
<h1 v-if="ok">Yes</h1>
```

也可以用 `v-else` 添加一个 “else” 块：

```
<h1 v-if="ok">Yes</h1>
<h1 v-else>No</h1>
```

template v-if

因为 `v-if` 是一个指令，需要将它添加到一个元素上。但是如果我们想切换多个元素呢？此时我们可以把一个 `<template>` 元素当做包装元素，并在上面使用 `v-if`，最终的渲染结果不会包含它。

```
<template v-if="ok">
 <h1>Title</h1>
 <p>Paragraph 1</p>
 <p>Paragraph 2</p>
</template>
```

v-show

另一个根据条件展示元素的选项是 `v-show` 指令。用法大体上一样：

```
<h1 v-show="ok">Hello!</h1>
```

不同的是有 `v-show` 的元素会始终渲染并保持在 DOM 中。 `v-show` 是简单的切换元素的 CSS 属性 `display` 。

注意 `v-show` 不支持 `<template>` 语法。

v-else

可以用 `v-else` 指令给 `v-if` 或 `v-show` 添加一个 “else 块”：

```
<div v-if="Math.random() > 0.5">
 Sorry
</div>
<div v-else>
 Not sorry
</div>
```

`v-else` 元素必须立即跟在 `v-if` 或 `v-show` 元素的后面——否则它不能被识别。

v-if vs. v-show

在切换 `v-if` 块时，Vue.js 有一个局部编译/卸载过程，因为 `v-if` 之中的模板也可能包括数据绑定或子组件。 `v-if` 是真实的条件渲染，因为它会确保条件块在切换当中合适地销毁与重建条件块内的事件监听器和子组件。

`v-if` 也是惰性的：如果在初始渲染时条件为假，则什么也不做——在条件第一次变为真时才开始局部编译（编译会被缓存起来）。

相比之下， `v-show` 简单得多——元素始终被编译并保留，只是简单地基于 CSS 切换。

一般来说， `v-if` 有更高的切换消耗而 `v-show` 有更高的初始渲染消耗。因此，如果需要频繁切换 `v-show` 较好，如果在运行时条件不大可能改变 `v-if` 较好。

列表渲染

v-for

可以使用 `v-for` 指令基于一个数组渲染一个列表。这个指令使用特殊的语法，形式为 `item in items` ， `items` 是数据数组， `item` 是当前数组元素的别名：

示例：

```
<ul id="example-1">
 <li v-for="item in items">
 {{ item.message }}
 </li>
</ul>
```

```
var example1 = new Vue({
 el: '#example-1',
 data: {
 items: [
 { message: 'Foo' },
 { message: 'Bar' }
 ]
 }
})
```

结果：

- Foo
- Bar

在 `v-for` 块内我们能完全访问父组件作用域内的属性，另有一个特殊变量 `$index` ，正如你猜到的，它是当前数组元素的索引：

```
<ul id="example-2">
 <li v-for="item in items">
 {{ parentMessage }} - {{ $index }} - {{ item.message }}
 </li>
</ul>
```

```
var example2 = new Vue({
 el: '#example-2',
 data: {
 parentMessage: 'Parent',
 items: [
 { message: 'Foo' },
 { message: 'Bar' }
 ]
 }
})
```

结果：

- Parent - 0 - Foo
- Parent - 1 - Bar

另外，你可以为索引指定一个别名（如果 `v-for` 用于一个对象，则可以为对象的键指定一个别名）：

```
<div v-for="(index, item) in items">
 {{ index }} {{ item.message }}
</div>
```

template v-for

类似于 template `v-if`，也可以将 `v-for` 用在 `<template>` 标签上，以渲染一个包含多个元素的块。例如：

```
<ul>
 <template v-for="item in items">
 <li>{{ item.msg }}</li>
 <li class="divider"></li>
 </template>
</ul>
```

数组变动检测

变异方法

Vue.js 包装了被观察数组的变异方法，故它们能触发视图更新。被包装的方法有：

- `push()`
- `pop()`
- `shift()`
- `unshift()`
- `splice()`
- `sort()`
- `reverse()`

你可以打开浏览器的控制台，用这些方法修改上例的 `items` 数组。例如： `example1.items.push({ message: 'Baz' })` 。

替换数组

变异方法，如名字所示，修改了原始数组。相比之下，也有非变异方法，如 `filter()` ， `concat()` 和 `slice()` ，不会修改原始数组而是返回一个新数组。在使用非变异方法时，可以直接用新数组替换旧数组：

```
example1.items = example1.items.filter(function (item) {
 return item.message.match(/Foo/)
})
```

可能你觉得这将导致 Vue.js 弃用已有 DOM 并重新渲染整个列表——幸运的是并非如此。 Vue.js 实现了一些启发算法，以最大化复用 DOM 元素，因而用另一个数组替换数组是一个非常高效的操作。

track-by

有时需要用全新对象（例如通过 API 调用创建的对象）替换数组。因为 `v-for` 默认通过数据对象的特征来决定对已有作用域和 DOM 元素的复用程度，这可能导致重新渲染整个列表。但是，如果每个对象都有一个唯一 ID 的属性，便可以使用 `track-by` 特性给 Vue.js 一个提示，Vue.js 因而能尽可能地复用已有实例。

例如，假定数据为：

```
{
 items: [
 { _uid: '88f869d', ... },
 { _uid: '7496c10', ... }
 ]
}
```

然后可以这样给出提示：

```
<div v-for="item in items" track-by="_uid">
 <!-- content -->
</div>
```

然后在替换数组 `items` 时，如果 Vue.js 遇到一个包含 `_uid: '88f869d'` 的新对象，它知道它可以复用这个已有对象的作用域与 DOM 元素。

track-by $index

如果没有唯一的键供追踪，可以使用 `track-by="$index"` ，它强制让 `v-for` 进入原位更新模式：片断不会被移动，而是简单地以对应索引的新值刷新。这种模式也能处理数据数组中重复的值。

这让数据替换非常高效，但是也会付出一定的代价。因为这时 DOM 节点不再映射数组元素顺序的改变，不能同步临时状态（比如 `<input>` 元素的值）以及组件的私有状态。因此，如果 `v-for` 块包含 `<input>` 元素或子组件，要小心使用 `track-by="$index"`

问题

因为 JavaScript 的限制，Vue.js 不能检测到下面数组变化：

1. 直接用索引设置元素，如 `vm.items[0] = {}` ；
2. 修改数据的长度，如 `vm.items.length = 0` 。

为了解决问题 (1)，Vue.js 扩展了观察数组，为它添加了一个 `$set()` 方法：

```
// 与 `example1.items[0] = ...` 相同，但是能触发视图更新
example1.items.$set(0, { childMsg: 'Changed!'})
```

至于问题 (2)，只需用一个空数组替换 `items` 。

除了 `$set()` ， Vue.js 也为观察数组添加了 `$remove()` 方法，用于从目标数组中查找并删除元素，在内部它调用 `splice()` 。因此，不必这样：

```
var index = this.items.indexOf(item)
if (index !== -1) {
 this.items.splice(index, 1)
}
```

只用这样：

```
this.items.$remove(item)
```

对象 v-for

也可以使用 `v-for` 遍历对象。除了 `$index` 之外，作用域内还可以访问另外一个特殊变量 `$key` 。

```
<ul id="repeat-object" class="demo">
 <li v-for="value in object">
 {{ $key }} : {{ value }}
 </li>
</ul>
```

```
new Vue({
 el: '#repeat-object',
 data: {
 object: {
 FirstName: 'John',
 LastName: 'Doe',
 Age: 30
 }
 }
})
```

结果：

- FirstName : John
- LastName : Doe
- Age : 30

也可以给对象的键提供一个别名：

```
<div v-for="(key, val) in object">
 {{ key }} {{ val }}
</div>
```

在遍历对象时，是按 `Object.keys()` 的结果遍历，但是不能保证它的结果在不同的 JavaScript 引擎下是一致的。

值域 v-for

`v-for` 也可以接收一个整数，此时它将重复模板数次。

```
<div>
 <span v-for="n in 10">{{ n }} </span>
</div>
```

结果：

0 1 2 3 4 5 6 7 8 9

显示过滤/排序的结果

有时我们想显示过滤/排序过的数组，同时不实际修改或重置原始数据。有两个办法：

1. 创建一个计算属性，返回过滤/排序过的数组；
2. 使用内置的过滤器 `filterBy` 和 `orderBy` 。

计算属性有更好的控制力，也更灵活，因为它是全功能 JavaScript。但是通常过滤器更方便，详细见 API。

方法与事件处理器

方法处理器

可以用 `v-on` 指令监听 DOM 事件：

```
<div id="example">
 <button v-on:click="greet">Greet</button>
</div>
```

我们绑定了一个单击事件处理器到一个方法 `greet` 。下面在 Vue 实例中定义这个方法：

```
var vm = new Vue({
 el: '#example',
 data: {
 name: 'Vue.js'
 },
 // 在 `methods` 对象中定义方法
 methods: {
 greet: function (event) {
 // 方法内 `this` 指向 vm
 alert('Hello ' + this.name + '!')
 // `event` 是原生 DOM 事件
 alert(event.target.tagName)
 }
 }
})

// 也可以在 JavaScript 代码中调用方法
vm.greet() // -> 'Hello Vue.js!'
```

自己测试一下：

Greet

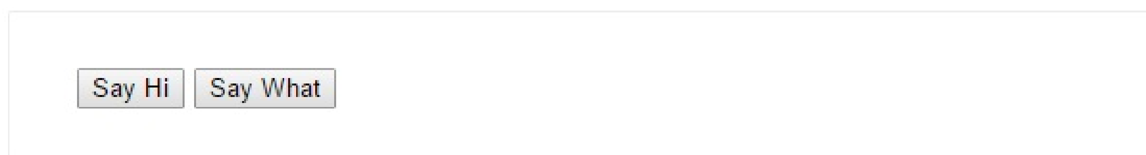
内联语句处理器

除了直接绑定到一个方法，也可以用内联 JavaScript 语句：

```
<div id="example-2">
 <button v-on:click="say('hi')">Say Hi</button>
 <button v-on:click="say('what')">Say What</button>
</div>
```

```
new Vue({
 el: '#example-2',
 methods: {
 say: function (msg) {
 alert(msg)
 }
 }
})
```

Result:



类似于内联表达式，事件处理器限制为一个语句。

有时也需要在内联语句处理器中访问原生 DOM 事件。可以用特殊变量 `$event` 把它传入方法：

```
<button v-on:click="say('hello!', $event)">Submit</button>
```

```
// ...
methods: {
 say: function (msg, event) {
 // 现在我们可以访问原生事件对象
 event.preventDefault()
 }
}
```

事件修饰符

在事件处理器中经常需要调用 `event.preventDefault()` 或 `event.stopPropagation()` 。尽管我们在方法内可以轻松做到，不过让方法是纯粹的数据逻辑而不处理 DOM 事件细节会更好。

为了解决这个问题，Vue.js 为 `v-on` 提供两个 事件修饰符： `.prevent` 与 `.stop` 。你是否还记得修饰符是点号打头的指令后缀？

```
<!-- 阻止单击事件冒泡 -->
<a v-on:click.stop="doThis"></a>

<!-- 提交事件不再重载页面 -->
<form v-on:submit.prevent="onSubmit"></form>

<!-- 修饰符可以串联 -->
<a v-on:click.stop.prevent="doThat">

<!-- 只有修饰符 -->
<form v-on:submit.prevent></form>
```

按键修饰符

在监听键盘事件时，我们经常需要检测 keyCode。Vue.js 允许为 `v-on` 添加按键修饰符：

```
<!-- 只有在 keyCode 是 13 时调用 vm.submit() -->
<input v-on:keyup.13="submit">
```

记住所有的 keyCode 比较困难，Vue.js 为最常用的按键提供别名：

```
<!-- 同上 -->
<input v-on:keyup.enter="submit">

<!-- 缩写语法 -->
<input @keyup.enter="submit">
```

全部的按键别名：

- enter
- tab
- delete
- esc
- space
- up
- down
- left
- right

另外 1.0.8+ 也支持单字母按键别名。

为什么在 HTML 中监听事件?

你可能注意到这种事件监听的方式违背了传统理念 “separation of concern”。不必担心，因为所有的 Vue.js 事件处理方法和表达式都严格绑定在当前视图的 ViewModel 上，它不会导致任何维护困难。实际上，使用 `v-on` 有几个好处：

1. 扫一眼 HTML 模板便能轻松定位在 JavaScript 代码里对应的方法。
2. 因为你无须在 JavaScript 里手动绑定事件，你的 ViewModel 代码可以是非常纯粹的逻辑，和 DOM 完全解耦，更易于测试。
3. 当一个 ViewModel 被销毁时，所有的事件处理器都会自动被删除。你无须担心如何自己清理它们。

表单控件绑定

基础用法

可以用 `v-model` 指令在表单控件元素上创建双向数据绑定。根据控件类型它自动选取正确的方法更新元素。尽管有点神奇， `v-model` 不过是语法糖，在用户输入事件中更新数据，以及特别处理一些极端例子。

Text

```
<span>Message is: {{ message }}</span>
<br>
<input type="text" v-model="message" placeholder="edit me">
```

Message is:

edit me

Checkbox

单个勾选框，逻辑值：

```
<input type="checkbox" id="checkbox" v-model="checked">
<label for="checkbox">{{ checked }}</label>
```

false

多个勾选框，绑定到同一个数组：

```
<input type="checkbox" id="jack" value="Jack" v-model="checkedNames
<label for="jack">Jack</label>
<input type="checkbox" id="john" value="John" v-model="checkedNames
<label for="john">John</label>
<input type="checkbox" id="mike" value="Mike" v-model="checkedNames
<label for="mike">Mike</label>
<br>
<span>Checked names: {{ checkedNames &#124; json }}</span>
```

☐ Jack ☐ John ☐ Mike
Checked names: []

Radio

```
<input type="radio" id="one" value="One" v-model="picked">
<label for="one">One</label>
<br>
<input type="radio" id="two" value="Two" v-model="picked">
<label for="two">Two</label>
<br>
<span>Picked: {{ picked }}</span>
```

○ One
○ Two
Picked:

Select

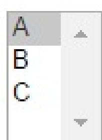
单选：

```
<select v-model="selected">
 <option selected>A</option>
 <option>B</option>
 <option>C</option>
</select>
<span>Selected: {{ selected }}</span>
```

A ▾ Selected: A

多选（绑定到一个数组）：

```
<select v-model="selected" multiple>
 <option selected>A</option>
 <option>B</option>
 <option>C</option>
</select>
<br>
<span>Selected: {{ selected &#124; json }}</span>
```



Selected: ["A"]

动态选项，用 `v-for` 渲染：

```
<select v-model="selected">
 <option v-for="option in options" v-bind:value="option.value">
 {{ option.text }}
 </option>
</select>
<span>Selected: {{ selected }}</span>
```

```
new Vue({
 el: '...',
 data: {
 selected: 'A',
 options: [
 { text: 'One', value: 'A' },
 { text: 'Two', value: 'B' },
 { text: 'Three', value: 'C' }
 ]
 }
})
```

One ▾ Selected: A

值绑定

对于单选按钮，勾选框及选择框选项， `v-model` 绑定的值通常是静态字符串（对于勾选框是逻辑值）：

```
<!-- 当选中时，`picked` 为字符串 "a" -->
<input type="radio" v-model="picked" value="a">

<!-- `toggle` 为 true 或 false -->
<input type="checkbox" v-model="toggle">

<!-- 当选中时，`selected` 为字符串 "abc" -->
<select v-model="selected">
 <option value="abc">ABC</option>
</select>
```

但是有时我们想绑定值到 Vue 实例一个动态属性上。可以用 `v-bind` 做到。 而且 `v-bind` 允许绑定输入框的值到非字符串值。

Checkbox

```
<input
 type="checkbox"
 v-model="toggle"
 v-bind:true-value="a"
 v-bind:false-value="b">
```

```
// 选中
vm.toggle === vm.a
// 取消选中
vm.toggle === vm.b
```

Radio

```
<input type="radio" v-model="pick" v-bind:value="a">
```

```
// 选中
vm.pick === vm.a
```

Select Options

```
<select v-model="selected">
  <!-- 对象字面量 -->
  <option v-bind:value="{ number: 123 }">123</option>
</select>
```

```
// 选中
typeof vm.selected // -> 'object'
vm.selected.number // -> 123
```

参数特性

lazy

在默认情况下， `v-model` 在 `input` 事件中同步输入框值与数据，可以添加一个特性 `lazy` ，从而改到在 `change` 事件中同步：

```
<!-- 在 "change" 而不是 "input" 事件中更新 -->
<input v-model="msg" lazy>
```

number

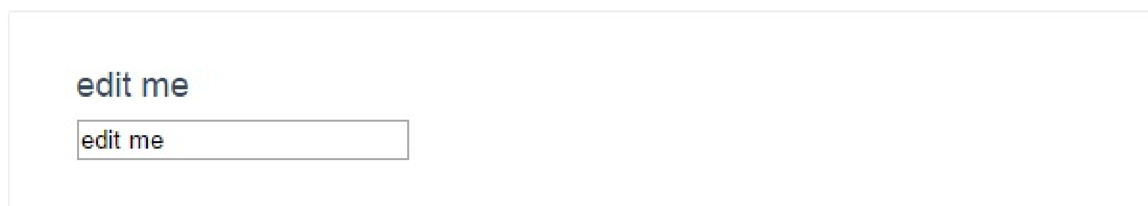
如果想自动将用户的输入保持为数字，可以添加一个特性 `number` ：

```
<input v-model="age" number>
```

debounce

`debounce` 设置一个最小的延时，在每次敲击之后延时同步输入框的值与数据。如果每次更新都要进行高耗操作（例如在输入提示中 Ajax 请求），它较为有用。

```
<input v-model="msg" debounce="500">
```



注意 `debounce` 参数不会延迟 input 事件：它延迟“写入”底层数据。因此在使用 `debounce` 时应当用 `vm.$watch()` 响应数据的变化。若想延迟 DOM 事件，应当使用 debounce 过滤器。

过渡

通过 Vue.js 的过渡系统，可以在元素从 DOM 中插入或移除时自动应用过渡效果。Vue.js 会在适当的时机为你触发 CSS 过渡或动画，你也可以提供相应的 JavaScript 钩子函数在过渡过程中执行自定义的 DOM 操作。

为了应用过渡效果，需要在目标元素上使用 `transition` 特性：

```
<div v-if="show" transition="my-transition"></div>
```

`transition` 特性可以与下面资源一起用：

- `v-if`
- `v-show`
- `v-for` （只为插入和删除触发）
- 动态组件 （介绍见组件）
- 在组件的根节点上，并且被 Vue 实例 DOM 方法（如 `vm.$appendTo(el)` ）触发。

当插入或删除带有过渡的元素时，Vue 将：

1. 尝试以 ID `"my-transition"` 查找 JavaScript 过渡钩子对象——通过 `Vue.transition(id, hooks)` 或 `transitions` 选项注册。如果找到了，将在过渡的不同阶段调用相应的钩子。

2. 自动嗅探目标元素是否有 CSS 过渡或动画，并在合适时添加/删除 CSS 类名。

3. 如果没有找到 JavaScript 钩子并且也没有检测到 CSS 过渡/动画，DOM 操作（插入/删除）在下一帧中立即执行。

CSS 过渡

示例

典型的 CSS 过渡像这样：

```
<div v-if="show" transition="expand">hello</div>
```

然后为 `.expand-transition` ， `.expand-enter` 和 `.expand-leave` 添加 CSS 规则:

```
/* 必需 */
.expand-transition {
 transition: all .3s ease;
 height: 30px;
 padding: 10px;
 background-color: #eee;
 overflow: hidden;
}

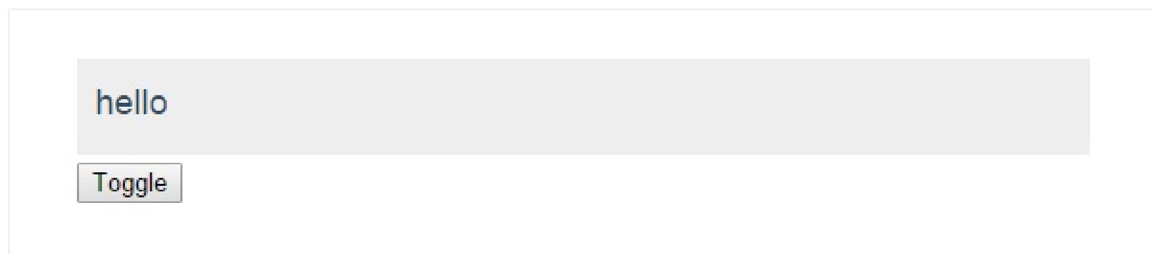
/* .expand-enter 定义进入的开始状态 */
/* .expand-leave 定义离开的结束状态 */
.expand-enter, .expand-leave {
 height: 0;
 padding: 0 10px;
 opacity: 0;
}
```

另外，可以提供 JavaScript 钩子:

```
Vue.transition('expand', {

 beforeEnter: function (el) {
 el.textContent = 'beforeEnter'
 },
 enter: function (el) {
 el.textContent = 'enter'
 },
 afterEnter: function (el) {
 el.textContent = 'afterEnter'
 },
 enterCancelled: function (el) {
 // handle cancellation
 },

 beforeLeave: function (el) {
 el.textContent = 'beforeLeave'
 },
 leave: function (el) {
 el.textContent = 'leave'
 },
 afterLeave: function (el) {
 el.textContent = 'afterLeave'
 },
 leaveCancelled: function (el) {
 // handle cancellation
 }
})
```



过渡的 CSS 类名

类名的添加和切换取决于 `transition` 特性的值。比如 `transition="fade"`，会有三个 CSS 类名：

1. `.fade-transition` 始终保留在元素上。
2. `.fade-enter` 定义进入过渡的开始状态。只应用一帧然后立即删除。
3. `.fade-leave` 定义离开过渡的结束状态。在离开过渡开始时生效，在它结束后删除。

如果 `transition` 特性没有值，类名默认是 `.v-transition`，`.v-enter` 和 `.v-leave`。

过渡流程详解

当 `show` 属性改变时，Vue.js 将相应地插入或删除 `<div>` 元素，按照如下规则改变过渡的 CSS 类名：

- 如果 `show` 变为 false，Vue.js 将：
    1. 调用 `beforeLeave` 钩子；
    2. 添加 `v-leave` 类名到元素上以触发过渡；
    3. 调用 `leave` 钩子；
    4. 等待过渡结束（监听 `transitionend` 事件）；
    5. 从 DOM 中删除元素并删除 `v-leave` 类名；
    6. 调用 `afterLeave` 钩子。
- 如果 `show` 变为 true，Vue.js 将：
    1. 调用 `beforeEnter` 钩子；
    2. 添加 `v-enter` 类名到元素上；
    3. 把它插入 DOM；
    4. 调用 `enter` 钩子；
    5. 强制一次 CSS 布局，让 `v-enter` 确实生效。然后删除 `v-enter` 类名，以触发过渡，回到元素的原始状态；
    6. 等待过渡结束；
    7. 调用 `afterEnter` 钩子。

另外，如果在它的进入过渡还在进行中时删除元素，将调用 `enterCancelled` 钩子，以清理变动或 `enter` 创建的计时器。反过来对于离开过渡亦如是。

上面所有的钩子函数在调用时，它们的 `this` 均指向所属的 Vue 实例。如果元素是 Vue 实例的根节点，则这个实例是上下文。否则，上下文是过渡指令所属的实例。

最后， `enter` 和 `leave` 可以有第二个可选的回调参数，用于显式控制过渡如何结束。因此不必等待 CSS `transitionend` 事件， Vue.js 将等待你手工调用这个回调，以结束过渡。例如：

```
enter: function (el) {
  // 没有第二个参数
  // 由 CSS transitionend 事件决定过渡何时结束
}
```

vs.

```
enter: function (el, done) {
  // 有第二个参数
  // 过渡只有在调用 `done` 时结束
}
```

当多个元素一起过渡时，Vue.js 会批量处理，只强制一次布局。

CSS 动画

CSS 动画用法同 CSS 过渡，区别是在动画中 `v-enter` 类名在节点插入 DOM 后不会立即删除，而是在 `animationend` 事件触发时删除。

示例：(省略了兼容性前缀)

```
<span v-show="show" transition="bounce">Look at me!</span>
```

```
.bounce-enter {
 animation: bounce-in .5s;
}
.bounce-leave {
 animation: bounce-out .5s;
}
@keyframes bounce-in {
 0% {
 transform: scale(0);
 }
 50% {
 transform: scale(1.5);
 }
 100% {
 transform: scale(1);
 }
}
@keyframes bounce-out {
 0% {
 transform: scale(1);
 }
 50% {
 transform: scale(1.5);
 }
 100% {
 transform: scale(0);
 }
}
```

Look at me!

Toggle

JavaScript 过渡

也可以只使用 JavaScript 钩子，不用定义任何 CSS 规则。当只使用 JavaScript 过渡时， `enter` 和 `leave` 钩子需要调用 `done` 回调，否则它们将被同步调用，过渡将立即结束。

为 JavaScript 过渡显式声明 `css: false` 是个好主意，Vue.js 将跳过 CSS 检测。这样也会阻止无意间让 CSS 规则干扰过渡。

在下例中我们使用 jQuery 注册一个自定义的 JavaScript 过渡：

```
Vue.transition('fade', {
 css: false,
 enter: function (el, done) {
 // 元素已被插入 DOM
 // 在动画结束后调用 done
 $(el)
 .css('opacity', 0)
 .animate({ opacity: 1 }, 1000, done)
 },
 enterCancelled: function (el) {
 $(el).stop()
 },
 leave: function (el, done) {
 // 与 enter 相同
 $(el).animate({ opacity: 0 }, 1000, done)
 },
 leaveCancelled: function (el) {
 $(el).stop()
 }
})
```

然后用 `transition` 特性中：

```
<p transition="fade"></p>
```

渐近过渡

`transition` 与 `v-for` 一起用时可以创建渐近过渡。给过渡元素添加一个特性 `stagger` , `enter-stagger` 或 `leave-stagger` ：

```
<div v-for="list" transition stagger="100"></div>
```

或者，提供一个钩子 `stagger` , `enter-stagger` 或 `leave-stagger` ，以更好的控制：

```
Vue.transition('stagger', {
 stagger: function (index) {
 // 每个过渡项目增加 50ms 延时
 // 但是最大延时限制为 300ms
 return Math.min(300, index * 50)
 }
})
```

示例：

Bruce Lee
Jackie Chan
Chuck Norris
Jet Li
Kung Fury

html

```
<div id="demo">
    <input v-model="query">
    <ul>
        <li v-for="item in list filterBy query"
            transition="staggered"
            stagger="100">
          {{item.msg}}
         </li>
    </ul>
</div>
```

js

```
new Vue({
    el: '#demo',
    data: {
        query: '',
        list: [
            { msg: 'Bruce Lee' },
            { msg: 'Jackie Chan' },
            { msg: 'Chuck Norris' },
            { msg: 'Jet Li' },
            { msg: 'Kung Fury' }
        ]
    }
})
```

css

```
ul {
    padding-left: 0;
    font-family: Helvetica, Arial, sans-serif;
}
.staggered-transition {
    transition: all .5s ease;
    overflow: hidden;
    margin: 0;
    height: 20px;
}
.staggered-enter, .staggered-leave {
    opacity: 0;
    height: 0;
}
```

组件

使用组件

注册

之前说过，我们可以用 `Vue.extend()` 创建一个组件构造器：

```
var MyComponent = Vue.extend({
 // 选项...
})
```

要把这个构造器用作组件，需要用 `Vue.component(tag, constructor)` 注册：

```
// 全局注册组件，tag 为 my-component
Vue.component('my-component', MyComponent)
```

对于自定义标签名字，Vue.js 不强制要求遵循 W3C 规则（小写，并且包含一个短杠），尽管遵循这个规则比较好。

在注册之后，组件便可以用在父实例的模块中，以自定义元素 `<my-component>` 的形式使用。要确保在初始化根实例之前注册了组件：

```
<div id="example">
 <my-component></my-component>
</div>
```

```
// 定义
var MyComponent = Vue.extend({
 template: '<div>A custom component!</div>'
})

// 注册
Vue.component('my-component', MyComponent)

// 创建根实例
new Vue({
 el: '#example'
})
```

渲染为：

```
<div id="example">
 <div>A custom component!</div>
</div>
```

A custom component!

注意组件的模板替换了自定义元素，自定义元素的作用只是作为一个挂载点。这可以用实例选项 `replace` 改变。

局部注册

不需要全局注册每个组件。可以让组件只能用在其它组件内，用实例选项 `components` 注册：

```
var Child = Vue.extend({ /* ... */ })

var Parent = Vue.extend({
 template: '...',
 components: {
 // <my-component> 只能用在父组件模板内
 'my-component': Child
 }
})
```

这种封装也适用于其它资源，如指令、过滤器和过渡。

注册语法糖

为了让事件更简单，可以直接传入选项对象而不是构造器给 `Vue.component()` 和 `component` 选项。Vue.js 在背后自动调用 `Vue.extend()` ：

```
// 在一个步骤中扩展与注册
Vue.component('my-component', {
 template: '<div>A custom component!</div>'
})

// 局部注册也可以这么做
var Parent = Vue.extend({
 components: {
 'my-component': {
 template: '<div>A custom component!</div>'
 }
 }
})
```

组件选项问题

传入 Vue 构造器的多数选项也可以用在 `Vue.extend()` 中，不过有两个特例： `data` and `el` 。试想如果我们简单地把一个对象作为 `data` 选项传给 `Vue.extend()` ：

```
var data = { a: 1 }
var MyComponent = Vue.extend({
 data: data
})
```

这么做的问题是 `MyComponent` 所有的实例将共享同一个 `data` 对象！这基本不是我们想要的，因此我们应当使用一个函数作为 `data` 选项，函数返回一个新对象：

```
var MyComponent = Vue.extend({
 data: function () {
 return { a: 1 }
 }
})
```

同理， `el` 选项用在 `Vue.extend()` 中时也须是一个函数。

`is` 特性

一些 HTML 元素，如 `<table>` ，限制什么元素可以放在它里面。自定义元素不在白名单上，将被放在元素的外面，因而渲染不正确。这时应当使用 `is` 特性，指示它是一个自定义元素：

```
<table>
 <tr is="my-component"></tr>
</table>
```

Props

使用 Props 传递数据

组件实例的作用域是孤立的。这意味着不能并且不应该在子组件的模板内直接引用父组件的数据。可以使用 **props** 把数据传给子组件。

“prop” 是组件数据的一个字段，期望从父组件传下来。子组件需要显式地用 `props` 选项 声明 props：

```
Vue.component('child', {
 // 声明 props
 props: ['msg'],
 // prop 可以用在模板内
 // 可以用 `this.msg` 设置
 template: '<span>{{ msg }}</span>'
})
```

然后向它传入一个普通字符串：

```
<child msg="hello!"></child>
```

结果：

hello!

camelCase vs. kebab-case

HTML 特性不区分大小写。名字形式为 camelCase 的 prop 用作特性时，需要转为 kebab-case（短横线隔开）：

```
Vue.component('child', {
 // camelCase in JavaScript
 props: ['myMessage'],
 template: '<span>{{ myMessage }}</span>'
})
```

```
<!-- kebab-case in HTML -->
<child my-message="hello!"></child>
```

动态 Props

类似于绑定一个普通的特性到一个表达式，也可以用 `v-bind` 绑定动态 Props 到父组件的数据。每当父组件的数据变化时，也会传导给子组件：

```
<div>
 <input v-model="parentMsg">
 <br>
 <child v-bind:my-message="parentMsg"></child>
</div>
```

使用 `v-bind` 的缩写语法通常更简单：

```
<child :my-message="parentMsg"></child>
```

结果：

Message from parent

Message from parent

字面量语法 vs. 动态语法

初学者常犯的一个错误是使用字面量语法传递数值：

```
<!-- 传递了一个字符串 "1" -->
<comp some-prop="1"></comp>
```

因为它是一个字面 prop，它的值以字符串 `"1"` 而不是以实际的数字传下去。如果想传递一个实际的 JavaScript 数字，需要使用动态语法，从而让它的值被当作 JavaScript 表达式计算：

```
<!-- 传递实际的数字 -->
<comp :some-prop="1"></comp>
```

Prop 绑定类型

prop 默认是单向绑定：当父组件的属性变化时，将传导给子组件，但是反过来不会。这是为了防止子组件无意修改了父组件的状态——这会让应用的数据流难以理解。不过，也可以使用 `.sync` 或 `.once` 绑定修饰符显式地强制双向或单次绑定：

比较语法：

```
<!-- 默认为单向绑定 -->
<child :msg="parentMsg"></child>

<!-- 双向绑定 -->
<child :msg.sync="parentMsg"></child>

<!-- 单次绑定 -->
<child :msg.once="parentMsg"></child>
```

双向绑定会把子组件的 `msg` 属性同步回父组件的 `parentMsg` 属性。单次绑定在建立之后不会同步之后的变化。

注意如果 prop 是一个对象或数组，是按引用传递。在子组件内修改它会影响父组件的状态，不管是使用哪种绑定类型。

Prop 验证

组件可以为 props 指定验证要求。当组件给其他人使用时这很有用，因为这些验证要求构成了组件的 API，确保其他人正确地使用组件。此时 props 的值是一个对象，包含验证要求：

```
Vue.component('example', {
 props: {
 // 基础类型检测 （`null` 意思是任何类型都可以）
 propA: Number,
 // 必需且是字符串
 propB: {
 type: String,
 required: true
 },
 // 数字，有默认值
 propC: {
 type: Number,
 default: 100
 },
 // 对象/数组的默认值应当由一个函数返回
 propD: {
 type: Object,
 default: function () {
 return { msg: 'hello' }
 }
 },
 // 指定这个 prop 为双向绑定
 // 如果绑定类型不对将抛出一条警告
 propE: {
 twoWay: true
 },
 // 自定义验证函数
 propF: {
 validator: function (value) {
 return value > 10
 }
 },
 // 转换函数（1.0.12 新增）
 // 在设置值之前转换值
 propG: {
 coerce: function (val) {
 return val + '' // 将值转换为字符串
 }
 }
 }
})
```

`type` 可以是下面原生构造器：

- String
- Number
- Boolean
- Function
- Object
- Array

`type` 也可以是一个自定义构造器，使用 `instanceof` 检测。

当 prop 验证失败了，Vue 将拒绝在子组件上设置此值，如果使用的是开发版本会抛出一条警告。

父子组件通信

父链

子组件可以用 `this.$parent` 访问它的父组件。根实例的后代可以用 `this.$root` 访问它。父组件有一个数组 `this.$children`，包含它所有的子元素。

尽管可以访问父链上任意的实例，不过子组件应当避免直接依赖父组件的数据，应当显式地使用 props 传递数据。另外，在子组件中修改父组件的状态是非常糟糕的做法，因为：

1. 这让父组件与子组件紧密地耦合；
2. 只看父组件，很难理解父组件的状态。因为它可能被任意子组件修改！理想情况下，只有组件自己能修改它的状态。

自定义事件

Vue 实例实现了一个自定义事件接口，用于在组件树中通信。这个事件系统独立于原生 DOM 事件，做法也不同。

每个 Vue 实例都是一个事件触发器：

- 使用 `$on()` 监听事件；
- 使用 `$emit()` 在它上面触发事件；
- 使用 `$dispatch()` 派发事件，事件沿着父链冒泡；
- 使用 `$broadcast()` 广播事件，事件向下传导给所有的后代。

不同于 DOM 事件，Vue 事件在冒泡过程中第一次触发回调之后自动停止冒泡，除非回调明确返回 `true` 。

简单例子：

```
<!-- 子组件模板 -->
<template id="child-template">
 <input v-model="msg">
 <button v-on:click="notify">Dispatch Event</button>
</template>

<!-- 父组件模板 -->
<div id="events-example">
 <p>Messages: {{ messages &#124; json }}</p>
 <child></child>
</div>
```

```
// 注册子组件
// 将当前消息派发出去
Vue.component('child', {
 template: '#child-template',
 data: function () {
 return { msg: 'hello' }
 },
 methods: {
 notify: function () {
 if (this.msg.trim()) {
 this.$dispatch('child-msg', this.msg)
 this.msg = ''
 }
 }
 }
})

// 启动父组件
// 将收到消息时将事件推入一个数组
var parent = new Vue({
 el: '#events-example',
 data: {
 messages: []
 },
 // 在创建实例时 `events` 选项简单地调用 `$on`
 events: {
 'child-msg': function (msg) {
 // 事件回调内的 `this` 自动绑定到注册它的实例上
 this.messages.push(msg)
 }
 }
})
```



使用 v-on 绑定自定义事件

上例非常好，不过看着父组件的代码， `"child-msg"` 事件来自哪里不直观。如果我们在模板中子组件用到的地方声明事件处理器会更好。为了做到这点，子组件可以用 `v-on` 监听自定义事件：

```
<child v-on:child-msg="handleIt"></child>
```

这让事情非常清晰：当子组件触发了 `"child-msg"` 事件，父组件的 `handleIt` 方法将被调用。所有影响父组件状态的代码放到父组件的 `handleIt` 方法中；子组件只关注触发事件。

子组件索引

尽管有 props 和 events，但是有时仍然需要在 JavaScript 中直接访问子组件。为此可以使用 `v-ref` 为子组件指定一个索引 ID。例如：

```
<div id="parent">
  <user-profile v-ref:profile></user-profile>
</div>
```

```
var parent = new Vue({ el: '#parent' })
// 访问子组件
var child = parent.$refs.profile
```

`v-ref` 和 `v-for` 一起用时，ref 是一个数组或对象，包含相应的子组件。

使用 Slot 分发内容

在使用组件时，常常要像这样组合它们：

```
<app>
  <app-header></app-header>
  <app-footer></app-footer>
</app>
```

注意两点：

1. `<app>` 组件不知道它的挂载点会有什么内容，挂载点的内容是由 `<app>` 的父组件决定的。
2. `<app>` 组件很可能有它自己的模板。

为了让组件可以组合，我们需要一种方式来混合父组件的内容与子组件自己的模板。这个处理称为内容分发（或 “transclusion”，如果你熟悉 Angular）。Vue.js 实现了一个内容分发 API，参照了当前 Web 组件规范草稿，使用特殊的 `<slot>` 元素作为原始内容的插槽。

编译作用域

在深入内容分发 API 之前，我们先明确内容的编译作用域。假定模板为：

```
<child>
  {{ msg }}
</child>
```

`msg` 应该绑定到父组件的数据，还是绑定到子组件的数据？答案是父组件。组件作用域简单地说是：

> 父组件模板的内容在父组件作用域内编译；子组件模板的内容在子组件作用域内编译

一个常见错误是试图在父组件模板内将一个指令绑定到子组件属性/方法：

```
<!-- 无效 -->
<child v-show="someChildProperty"></child>
```

假定 `someChildProperty` 是子组件的属性，上例不能如预期工作。父组件模板不知道子组件的状态。

如果要绑定子组件内的指令到一个组件的根节点，应当在它的模板内这么做：

```
Vue.component('child-component', {
 // 有效，因为是在正确的作用域内
 template: '<div v-show="someChildProperty">Child</div>',
 data: function () {
 return {
 someChildProperty: true
 }
 }
})
```

类似地，分发内容是在父组件作用域内编译。

单个 Slot

父组件的内容将被抛弃，除非子组件模板包含 `<slot>` 。如果只有一个没有特性的 slot，整个内容将被插到它所在的地方，替换 slot。

`<slot>` 标签的内容视为回退内容。回退内容在子组件的作用域内编译，只有当宿主元素为空并且没有内容供插入时显示。

假定 `my-component` 组件有下面模板：

```
<div>
 <h1>This is my component!</h1>
 <slot>
 This will only be displayed if there is no content
 to be distributed.
 </slot>
</div>
```

父组件模板：

```
<my-component>
 <p>This is some original content</p>
 <p>This is some more original content</p>
</my-component>
```

渲染结果：

```
<div>
 <h1>This is my component!</h1>
 <p>This is some original content</p>
 <p>This is some more original content</p>
</div>
```

命名 Slot

`<slot>` 元素有一个特殊特性 `name` ，用于配置如何分发内容。多个 slot 可以有不同的名字。命名 slot 将匹配有对应 `slot` 特性的内容片断。

也可以有一个未命名 slot，它是默认 **slot**，作为找不到匹配内容的回退插槽。如果没有默认的 slot，不匹配内容将被抛弃。

例如，假定我们有一个 `multi-insertion` 组件，它的模板为：

```
<div>
 <slot name="one"></slot>
 <slot></slot>
 <slot name="two"></slot>
</div>
```

父组件模板：

```
<multi-insertion>
 <p slot="one">One</p>
 <p slot="two">Two</p>
 <p>Default A</p>
</multi-insertion>
```

渲染结果为：

```
<div>
 <p slot="one">One</p>
 <p>Default A</p>
 <p slot="two">Two</p>
</div>
```

在组合组件时，内容分发 API 是非常有用的机制。

动态组件

多个组件可以使用同一个挂载点，然后动态地在它们之间切换。使用保留的 `<component>` 元素，动态地绑定到它的 `is` 特性：

```
new Vue({
 el: 'body',
 data: {
 currentView: 'home'
 },
 components: {
 home: { /* ... */ },
 posts: { /* ... */ },
 archive: { /* ... */ }
 }
})
```

```
<component :is="currentView">
 <!-- 组件在 vm.currentview 变化时改变 -->
</component>
```

如果把切换出去的组件保留在内存中，可以保留它的状态或避免重新渲染。为此可以添加一个 `keep-alive` 指令参数：

```
<component :is="currentView" keep-alive>
 <!-- 非活动组件将被缓存 -->
</component>
```

`activate` 钩子

在切换组件时，切入组件在切入前可能需要进行一些异步操作。为了控制组件切换时长，给切入组件添加 `activate` 钩子：

```
Vue.component('activate-example', {
 activate: function (done) {
 var self = this
 loadDataAsync(function (data) {
 self.someData = data
 done()
 })
 }
})
```

注意 `activate` 钩子只作用于动态组件切换或静态组件初始化渲染的过程中，不作用于使用实例方法手工插入的过程中。

`transition-mode`

`transition-mode` 特性用于指定两个动态组件之间如何过渡。

在默认情况下，进入与离开平滑地过渡。这个特性可以指定另外两种模式：

- `in-out` ：新组件先过渡进入，等它的过渡完成之后当前组件过渡出去。
- `out-in` ：当前组件先过渡出去，等它的过渡完成之后新组件过渡进入。

示例：

```
<!-- 先淡出再淡入 -->
<component
 :is="view"
 transition="fade"
 transition-mode="out-in">
</component>
```

```
.fade-transition {
 transition: opacity .3s ease;
}
.fade-enter, .fade-leave {
 opacity: 0;
}
```

A B
Component A

杂项

组件和 v-for

自定义组件可以像普通元素一样直接使用 `v-for` ：

```
<my-component v-for="item in items"></my-component>
```

但是，不能传递数据给组件，因为组件的作用域是孤立的。为了传递数据给组件，应当使用 props：

```
<my-component
 v-for="item in items"
 :item="item"
 :index="$index">
</my-component>
```

不自动把 `item` 注入组件的原因是这会导致组件跟当前 `v-for` 紧密耦合。显式声明数据来自哪里可以让组件复用在其它地方。

编写可复用组件

在编写组件时，记住是否要复用组件有好处。一次性组件跟其它组件紧密耦合没关系，但是可复用组件应当定义一个清晰的公开接口。

Vue.js 组件 API 来自三部分——prop，事件和 slot：

- **prop** 允许外部环境传递数据给组件；
- **事件** 允许组件触发外部环境的 action；
- **slot** 允许外部环境插入内容到组件的视图结构内。

使用 `v-bind` 和 `v-on` 的简写语法，模板的缩进清楚且简洁：

```
<my-component
 :foo="baz"
 :bar="qux"
 @event-a="doThis"
 @event-b="doThat">
 <!-- content -->
 <img slot="icon" src="...">
 <p slot="main-text">Hello!</p>
</my-component>
```

异步组件

在大型应用中，我们可能需要将应用拆分为小块，只在需要时才从服务器下载。为了让事情更简单，Vue.js 允许将组件定义为一个工厂函数，动态地解析组件的定义。Vue.js 只在组件需要渲染时触发工厂函数，并且把结果缓存起来，用于后面的再次渲染。例如：

```
Vue.component('async-example', function (resolve, reject) {
 setTimeout(function () {
 resolve({
 template: '<div>I am async!</div>'
 })
 }, 1000)
})
```

工厂函数接收一个 `resolve` 回调，在收到从服务器下载的组件定义时调用。也可以调用 `reject(reason)` 指示加载失败。这里 `setTimeout` 只是为了演示。怎么获取组件完全由你决定。推荐配合使用 Webpack 的代码分割功能：

```
Vue.component('async-webpack-example', function (resolve) {
 // 这个特殊的 require 语法告诉 webpack
 // 自动将编译后的代码分割成不同的块，
 // 这些块将通过 ajax 请求自动下载。
 require(['./my-async-component'], resolve)
})
```

资源命名约定

一些资源，如组件和指令，是以 HTML 特性或 HTML 自定义元素的形式出现在模板中。因为 HTML 特性的名字和标签的名字不区分大小写，所以资源的名字通常需使用 kebab-case 而不是 camelCase 的形式，这不大方便。

Vue.js 支持资源的名字使用 camelCase 或 PascalCase 的形式，并且在模板中自动将它们转为 kebab-case（类似于 prop 的命名约定）：

```
// 在组件定义中
components: {
 // 使用 camelCase 形式注册
 myComponent: { /*... */ }
}
```

```
<!-- 在模板中使用 kebab-case 形式 -->
<my-component></my-component>
```

ES6 对象字面量缩写 也没问题：

```
// PascalCase
import TextBox from './components/text-box';
import DropdownMenu from './components/dropdown-menu';

export default {
 components: {
 // 在模板中写作 <text-box> 和 <dropdown-menu>
 TextBox,
 DropdownMenu
 }
}
```

递归组件

组件在它的模板内可以递归地调用自己，不过，只有当它有 `name` 选项时才可以：

```
var StackOverflow = Vue.extend({
 name: 'stack-overflow',
 template:
 '<div>' +
 // 递归地调用它自己
 '<stack-overflow></stack-overflow>' +
 '</div>'
})
```

上面组件会导致一个错误 “max stack size exceeded”，所以要确保递归调用有终止条件。当使用 `Vue.component()` 全局注册一个组件时，组件 ID 自动设置为组件的 `name` 选项。

片断实例

在使用 `template` 选项时，模板的内容将替换实例的挂载元素。因而推荐模板的顶级元素始终是单个元素。

不这么写模板：

```
<div>root node 1</div>
<div>root node 2</div>
```

推荐这么写：

```
<div>
 I have a single root node!
 <div>node 1</div>
 <div>node 2</div>
</div>
```

下面几种情况会让实例变成一个片断实例：

1. 模板包含多个顶级元素。
2. 模板只包含普通文本。
3. 模板只包含其它组件（其它组件可能是一个片段实例）。
4. 模板只包含一个元素指令，如 `<partial>` 或 vue-router 的 `<router-view>` 。
5. 模板根节点有一个流程控制指令，如 `v-if` 或 `v-for` 。

这些情况让实例有未知数量的顶级元素，它将把它的 DOM 内容当作片断。片断实例仍然会正确地渲染内容。不过，它没有一个根节点，它的 `$el` 指向一个锚节点，即一个空的文本节点（在开发模式下是一个注释节点）。

但是更重要的是，组件元素上的非流程控制指令，非 **prop** 特性和过渡将被忽略，因为没有根元素供绑定：

```
<!-- 不可以，因为没有根元素 -->
<example v-show="ok" transition="fade"></example>

<!-- props 可以 -->
<example :prop="someData"></example>

<!-- 流程控制可以，但是不能有过渡 -->
<example v-if="ok"></example>
```

当然片断实例有它的用处，不过通常给组件一个根节点比较好。它会保证组件元素上的指令和特性能正确地转换，同时性能也稍微好些。

内联模板

如果子组件有 `inline-template` 特性，组件将把它的内容当作它的模板，而不是把它当作分发内容。这让模板更灵活。

```
<my-component inline-template>
 <p>These are compiled as the component's own template</p>
 <p>Not parent's transclusion content.</p>
</my-component>
```

但是 `inline-template` 让模板的作用域难以理解，并且不能缓存模板编译结果。最佳实践是使用 `template` 选项在组件内定义模板。

深入响应式原理

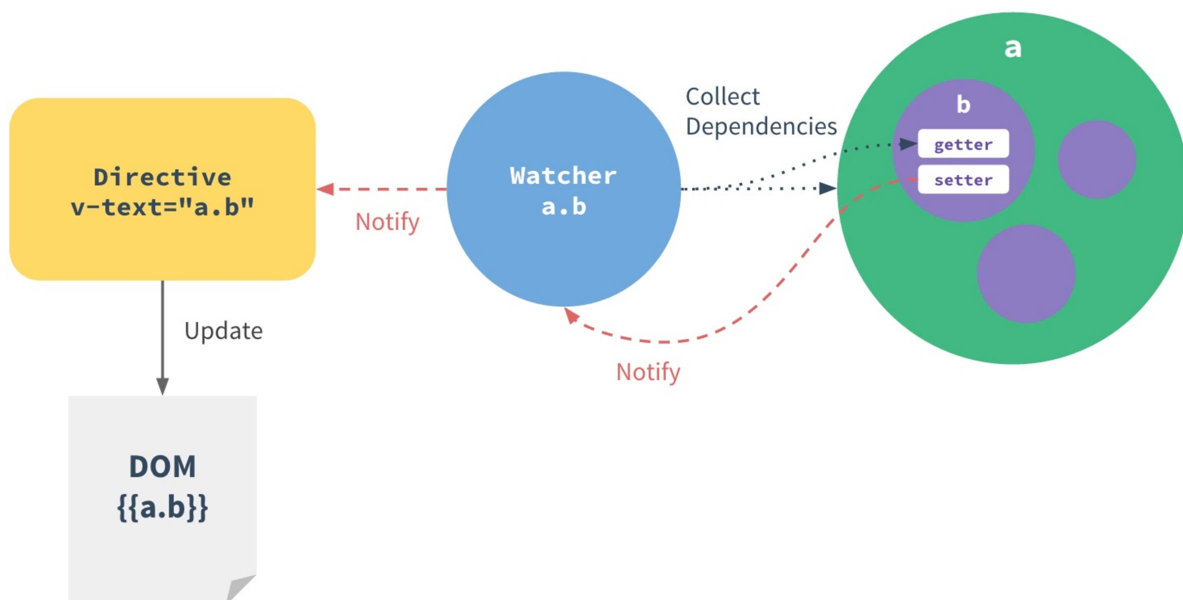
大部分的基础内容我们已经讲到了，现在讲点底层内容。Vue.js 最显著的一个功能是响应系统 —— 模型只是普通对象，修改它则更新视图。这让状态管理非常简单且直观，不过理解它的原理也很重要，可以避免一些常见问题。下面我们开始深挖 Vue.js 响应系统的底层细节。

如何追踪变化

把一个普通对象传给 Vue 实例作为它的 `data` 选项，Vue.js 将遍历它的属性，用 Object.defineProperty 将它们转为 getter/setter。这是 ES5 特性，不能打补丁实现，这便是为什么 Vue.js 不支持 IE8 及更低版本。

用户看不到 getter/setters，但是在内部它们让 Vue.js 追踪依赖，在属性被访问和修改时通知变化。一个问题是在浏览器控制台打印数据对象时 getter/setter 的格式化不同，使用 `vm.$log()` 实例方法可以得到更友好的输出。

模板中每个指令/数据绑定都有一个对应的 **watcher** 对象，在计算过程中它把属性记录为依赖。之后当依赖的 setter 被调用时，会触发 watcher 重新计算，也就会导致它的关联指令更新 DOM。



变化检测问题

受 ES5 的限制，Vue.js 不能检测到对象属性的添加或删除。因为 Vue.js 在初始化实例时将属性转为 getter/setter，所以属性必须在 `data` 对象上才能让 Vue.js 转换它，才能让它是响应的。例如：

```
var data = { a: 1 }
var vm = new Vue({
  data: data
})
// `vm.a` 和 `data.a` 现在是响应的

vm.b = 2
// `vm.b` 不是响应的

data.b = 2
// `data.b` 不是响应的
```

不过，有办法在实例创建之后添加属性并且让它是响应的。

对于 Vue 实例，可以使用 `$set(key, value)` 实例方法：

```
vm.$set('b', 2)
// `vm.b` 和 `data.b` 现在是响应的
```

对于普通数据对象，可以使用全局方法 `Vue.set(object, key, value)` ：

```
Vue.set(data, 'c', 3)
// `vm.c` 和 `data.c` 现在是响应的
```

有时你想向已有对象上添加一些属性，例如使用 `Object.assign()` 或 `_.extend()` 添加属性。但是，添加到对象上的新属性不会触发更新。这时可以创建一个新的对象，包含原对象的属性和新的属性：

```
// 不使用 `Object.assign(this.someObject, { a: 1, b: 2 })`
this.someObject = Object.assign({}, this.someObject, { a: 1, b: 2 
```

也有一些数组相关的问题，之前已经在列表渲染中讲过。

初始化数据

尽管 Vue.js 提供了 API 动态地添加响应属性，还是推荐在 `data` 对象上声明所有的响应属性。

不这么做：

```
var vm = new Vue({
 template: '<div>{{msg}}</div>'
})
// 然后添加 `msg`
vm.$set('msg', 'Hello!')
```

这么做：

```
var vm = new Vue({
 data: {
 // 以一个空值声明 `msg`
 msg: ''
 },
 template: '<div>{{msg}}</div>'
})
// 然后设置 `msg`
vm.msg = 'Hello!'
```

这么做有两个原因：

1. `data` 对象就像组件状态的模式（schema）。在它上面声明所有的属性让组件代码更易于理解。
2. 添加一个顶级响应属性会强制所有的 watcher 重新计算，因为它之前不存在，没有 watcher 追踪它。这么做性能通常是可以接受的（特别是对比 Angular 的脏检查），但是可以在初始化时避免。

异步更新队列

Vue.js 默认异步更新 DOM。每当观察到数据变化时，Vue 就开始一个队列，将同一事件循环内所有的数据变化缓存起来。如果一个 watcher 被多次触发，只会推入一次到队列中。等到下一次事件循环，Vue 将清空队列，只进行必要的 DOM 更新。在内部异步队列优先使用 `MutationObserver` ，如果不支持则使用 `setTimeout(fn, 0)` 。

例如，设置了 `vm.someData = 'new value'` ，DOM 不会立即更新，而是在下一次事件循环清空队列时更新。我们基本不用关心这个过程，但是如果想在 DOM 状态更新后做点什么，这会有帮助。尽管 Vue.js 鼓励开发者沿着数据驱动的思路，避免直接修改 DOM，但是有时确实要这么做。为了在数据变化之后等待 Vue.js 完成更新 DOM，可以在数据变化之后立即使用 `Vue.nextTick(callback)` 。回调在 DOM 更新完成后调用。例如：

```
<div id="example">{{msg}}</div>
```

```
var vm = new Vue({
 el: '#example',
 data: {
 msg: '123'
 }
})
vm.msg = 'new message' // 修改数据
vm.$el.textContent === 'new message' // false
Vue.nextTick(function () {
 vm.$el.textContent === 'new message' // true
})
```

`vm.$nextTick()` 这个实例方法比较方便，因为它不需要全局 `Vue` ，它的回调的 `this` 自动绑定到当前 Vue 实例：

```
Vue.component('example', {
 template: '<span>{{msg}}</span>',
 data: function () {
 return {
 msg: 'not updated'
 }
 },
 methods: {
 updateMessage: function () {
 this.msg = 'updated'
 console.log(this.$el.textContent) // => 'not updated'
 this.$nextTick(function () {
 console.log(this.$el.textContent) // => 'updated'
 })
 }
 }
})
```

计算属性的秘密

应注意到 Vue.js 的计算属性不是简单的 getter。计算属性持续追踪它的响应依赖。在计算一个计算属性时，Vue.js 更新它的依赖列表并缓存结果，只有当其中一个依赖发生了变化，缓存的结果才无效。因此，只要依赖不发生变化，访问计算属性会直接返回缓存的结果，而不是调用 getter。

为什么要缓存呢？假设我们有一个高耗计算属性 `A` ，它要遍历一个巨型数组并做大量的计算。然后，可能有其它的计算属性依赖 `A` 。如果没有缓存，我们将调用 `A` 的 getter 许多次，超过必要次数。

由于计算属性被缓存了，在访问它时 getter 不总是被调用。考虑下例：

```
var vm = new Vue({
 data: {
 msg: 'hi'
 },
 computed: {
 example: function () {
 return Date.now() + this.msg
 }
 }
})
```

计算属性 `example` 只有一个依赖：`vm.msg` 。`Date.now()` 不是响应依赖，因为它跟 Vue 的数据观察系统无关。因而，在访问 `vm.example` 时将发现时间戳不变，除非 `vm.msg` 变了。

有时希望 getter 不改变原有的行为，每次访问 `vm.example` 时都调用 getter。这时可以为指定的计算属性关闭缓存：

```
computed: {
 example: {
 cache: false,
 get: function () {
 return Date.now() + this.msg
 }
 }
}
```

现在每次访问 `vm.example` 时，时间戳都是新的。但是，只是在 **JavaScript** 中访问是这样的；数据绑定仍是依赖驱动的。如果在模块中这样绑定计算属性 `{{example}}` ，只有响应依赖发生变化时才更新 DOM。

自定义指令

基础

除了内置指令，Vue.js 也允许注册自定义指令。自定义指令提供一种机制将数据的变化映射为 DOM 行为。

可以用 `Vue.directive(id, definition)` 方法注册一个全局自定义指令，它接收两个参数指令 **ID** 与定义对象。也可以用组件的 `directives` 选项注册一个局部自定义指令。

钩子函数

定义对象可以提供几个钩子函数（都是可选的）：

- **bind**：只调用一次，在指令第一次绑定到元素上时调用。
- **update**： 在 `bind` 之后立即以初始值为参数第一次调用，之后每当绑定值变化时调用，参数为新值与旧值。
- **unbind**：只调用一次，在指令从元素上解绑时调用。

示例

```
Vue.directive('my-directive', {
 bind: function () {
 // 准备工作
 // 例如，添加事件处理器或只需要运行一次的高耗任务
 },
 update: function (newValue, oldValue) {
 // 值更新时的工作
 // 也会以初始值为参数调用一次
 },
 unbind: function () {
 // 清理工作
 // 例如，删除 bind() 添加的事件监听器
 }
})
```

在注册之后，便可以在 Vue.js 模板中这样用（记着添加前缀 `v-` ）：

```
<div v-my-directive="someValue"&gt;</div&gt;
```

当只需要 `update` 函数时，可以传入一个函数替代定义对象：

```
Vue.directive('my-directive', function (value) {
 // 这个函数用作 update()
})
```

指令实例属性

所有的钩子函数将被复制到实际的指令对象中，钩子内 `this` 指向这个指令对象。这个对象暴露了一些有用的属性：

- **el**: 指令绑定的元素。
- **vm**: 拥有该指令的上下文 ViewModel。
- **expression**: 指令的表达式，不包括参数和过滤器。
- **arg**: 指令的参数。
- **name**: 指令的名字，不包含前缀。
- **modifiers**: 一个对象，包含指令的修饰符。
- **descriptor**: 一个对象，包含指令的解析结果。

你应当将这些属性视为只读的，不要修改它们。你也可以给指令对象添加自定义属性，但是注意不要覆盖已有的内部属性。

示例：

```
<div id="demo" v-demo:hello.a.b="msg"&gt;</div&gt;
```

```
Vue.directive('demo', {
 bind: function () {
 console.log('demo bound!')
 },
 update: function (value) {
 this.el.innerHTML =
 'name - '       + this.name + '<br&gt;' +
 'expression - ' + this.expression + '<br&gt;' +
 'argument - '   + this.arg + '<br&gt;' +
 'modifiers - '  + JSON.stringify(this.modifiers) + '<br&gt;' +
 'value - '      + value
 }
})
var demo = new Vue({
 el: '#demo',
 data: {
 msg: 'hello!'
 }
})
```

结果

```
name - demo
expression - msg
argument - hello
modifiers - {"b":true,"a":true}
value - hello!
```

对象字面量

如果指令需要多个值，可以传入一个 JavaScript 对象字面量。记住，指令可以使用任意合法的 JavaScript 表达式：

```
<div v-demo="{ color: 'white', text: 'hello!' }"&gt;</div&gt;
```

```
Vue.directive('demo', function (value) {
 console.log(value.color) // "white"
 console.log(value.text) // "hello!"
})
```

字面修饰符

当指令使用了字面修饰符，它的值将按普通字符串处理并传递给 `update` 方法。 `update` 方法将只调用一次，因为普通字符串不能响应数据变化。

```
<div v-demo.literal="foo bar baz"&gt;
```

```
Vue.directive('demo', function (value) {
 console.log(value) // "foo bar baz"
})
```

元素指令

有时我们想以自定义元素的形式使用指令，而不是以特性的形式。这与 Angular 的 “E” 指令非常相似。元素指令可以看做是一个轻量组件。可以像下面这样注册一个自定义元素指令：

```
Vue.elementDirective('my-directive', {
 // API 同普通指令
 bind: function () {
 // 操作 this.el...
 }
})
```

不这样写：

```
<div v-my-directive&gt;</div&gt;
```

这样写：

```
<my-directive&gt;</my-directive&gt;
```

元素指令不能接受参数或表达式，但是它可以读取元素的特性从而决定它的行为。

迥异于普通指令，元素指令是终结性的，这意味着，一旦 Vue 遇到一个元素指令，它将跳过该元素及其子元素——只有该元素指令本身可以操作该元素及其子元素。

高级选项

params

自定义指令可以接收一个 `params` 数组，指定一个特性列表，Vue 编译器将自动提取绑定元素的这些特性。例如：

```
<div v-example a="hi"&gt;</div&gt;
```

```
Vue.directive('example', {
 params: ['a'],
 bind: function () {
 console.log(this.params.a) // -&gt; "hi"
 }
})
```

此 API 也支持动态属性。 `this.params[key]` 会自动保持更新。另外，可以指定一个回调，在值变化时调用：

```
<div v-example v-bind:a="someValue"&gt;</div&gt;
```

```
Vue.directive('example', {
 params: ['a'],
 paramWatchers: {
 a: function (val, oldVal) {
 console.log('a changed!')
 }
 }
})
```

deep

如果自定义指令用在一个对象上，当对象内部属性变化时要触发 `update` ，则在指令定义对象中指定 `deep: true` 。

```
<div v-my-directive="obj"&gt;</div&gt;
```

```
Vue.directive('my-directive', {
 deep: true,
 update: function (obj) {
 // 在 `obj` 的嵌套属性变化时调用
 }
})
```

twoWay

如果指令想向 Vue 实例写回数据，则在指令定义对象中指定 `twoWay: true` 。该选项允许在指令中使用 `this.set(value)` :

```
Vue.directive('example', {
 twoWay: true,
 bind: function () {
 this.handler = function () {
 // 将数据写回 vm
 // 如果指令这样绑定 v-example="a.b.c"
 // 它将用给定值设置 `vm.a.b.c`
 this.set(this.el.value)
 }.bind(this)
 this.el.addEventListener('input', this.handler)
 },
 unbind: function () {
 this.el.removeEventListener('input', this.handler)
 }
})
```

acceptStatement

传入 `acceptStatement:true` 可以让自定义指令接受内联语句，就像 `v-on` 那样：

```
<div v-my-directive="a++"&gt;</div&gt;
```

```
Vue.directive('my-directive', {
 acceptStatement: true,
 update: function (fn) {
 // 传入值是一个函数
 // 在调用它时将在所属实例作用域内计算 "a++" 语句
 }
})
```

明智地使用，因为通常你要在模板中避免副效应。

priority

可以给指令指定一个优先级（默认是 1000）。同一个元素上优先级高的指令会比其它指令处理得早一些。优先级一样的指令按照它在元素特性列表中出现的顺序依次处理，但是不能保证这个顺序在不同的浏览器中是一致的。

可以在 API 中查看内置指令的优先级。另外，流程控制指令 `v-if` 和 `v-for` 在编译过程中始终拥有最高的优先级。

自定义过滤器

基础

类似于自定义指令，可以用全局方法 `Vue.filter()` 注册一个自定义过滤器，它接收两个参数：过滤器 **ID** 和过滤器函数。过滤器函数以值为参数，返回转换后的值：

```
Vue.filter('reverse', function (value) {
 return value.split('').reverse().join('')
})
```

```
<!-- 'abc' => 'cba' -->
<span v-text="message &#124; reverse"></span>
```

过滤器函数可以接收任意数量的参数：

```
Vue.filter('wrap', function (value, begin, end) {
 return begin + value + end
})
```

```
<!-- 'hello' => 'before hello after' -->
<span v-text="message &#124; wrap 'before' 'after'"></span>
```

双向过滤器

目前我们使用过滤器都是在把来自模型的值显视在视图之前转换它。不过也可以定义一个过滤器，在把来自视图（ `<input>` 元素）的值写回模型之前转化它：

```
Vue.filter('currencyDisplay', {
 // model -> view
 // 在更新 `<input>` 元素之前格式化值
 read: function(val) {
 return '/figure>+val.toFixed(2)
 },
 // view -> model
 // 在写回数据之前格式化值
 write: function(val, oldVal) {
 var number = +val.replace(/[^\d.]/g, '')
 return isNaN(number) ? 0 : parseFloat(number.toFixed(2))
 }
})
```

示例：

$123.45

Model value: 123.45

动态参数

如果过滤器参数没有用引号包起来，则它会在当前 vm 作用域内动态计算。另外，过滤器函数的 `this` 始终指向调用它的 vm。例如：

```
<input v-model="userInput">
<span>{{msg &#124; concat userInput}}</span>
```

```
Vue.filter('concat', function (value, input) {
 // `input` === `this.userInput`
 return value + input
})
```

上例比较简单，也可以用表达式达到相同的结果，但是对于更复杂的逻辑——需要多于一个语句，这时需要将它放到计算属性或自定义过滤器中。

内置过滤器 `filterBy` 和 `orderBy`，根据所属 Vue 实例的当前状态，过滤/排序传入的数组。

混合

基础

混合以一种灵活的方式为组件提供分布复用功能。混合对象可以包含任意的组件选项。当组件使用了混合对象时，混合对象的所有选项将被“混入”组件自己的选项中。

示例：

```
// 定义一个混合对象
var myMixin = {
 created: function () {
 this.hello()
 },
 methods: {
 hello: function () {
 console.log('hello from mixin!')
 }
 }
}

// 定义一个组件，使用这个混合对象
var Component = Vue.extend({
 mixins: [myMixin]
})

var component = new Component() // -> "hello from mixin!"
```

选项合并

当混合对象与组件包含同名选项时，这些选项将以适当的策略合并。例如，同名钩子函数被并入一个数组，因而都会被调用。另外，混合的钩子将在组件自己的钩子之前调用。

```
var mixin = {
 created: function () {
 console.log('mixin hook called')
 }
}

new Vue({
 mixins: [mixin],
 created: function () {
 console.log('component hook called')
 }
})

// -> "mixin hook called"
// -> "component hook called"
```

值为对象的选项，如 `methods` ，`components` 和 `directives` 将合并到同一个对象内。如果键冲突则组件的选项优先。

```
var mixin = {
 methods: {
 foo: function () {
 console.log('foo')
 },
 conflicting: function () {
 console.log('from mixin')
 }
 }
}

var vm = new Vue({
 mixins: [mixin],
 methods: {
 bar: function () {
 console.log('bar')
 },
 conflicting: function () {
 console.log('from self')
 }
 }
})

vm.foo() // -> "foo"
vm.bar() // -> "bar"
vm.conflicting() // -> "from self"
```

注意 `Vue.extend()` 使用同样的合并策略。

全局混合

也可以全局注册混合。小心使用！一旦全局注册混合，它会影响所有之后创建的 Vue 实例。如果使用恰当，可以为自定义选项注入处理逻辑：

```
// 为 `myOption` 自定义选项注入一个处理器
Vue.mixin({
 created: function () {
 var myOption = this.$options.myOption
 if (myOption) {
 console.log(myOption)
 }
 }
})

new Vue({
 myOption: 'hello!'
})
// -> "hello!"
```

自定义选项合并策略

在合并自定义选项时，默认的合并策略是简单地覆盖已有值。如果想用自定义逻辑合并自定义选项，则向 `Vue.config.optionMergeStrategies` 添加一个函数：

```
Vue.config.optionMergeStrategies.myOption = function (toVal, fromVa
 // 返回 mergedVal
}
```

对于多数值为对象的选项，可以简单地使用 `methods` 所用的合并策略:

```
var strategies = Vue.config.optionMergeStrategies
strategies.myOption = strategies.methods
```

插件

开发插件

插件通常会为 Vue 添加全局功能。插件的范围没有限制——通常是下面几种：

1. 添加全局方法或属性，如 vue-element
2. 添加全局资源：指令/过滤器/过渡等，如 vue-touch
3. 添加 Vue 实例方法，通过把它们添加到 Vue.prototype 上实现。
4. 一个库，提供自己的 API，同时提供上面提到的一个或多个功能，如 vue-router

Vue.js 的插件应当有一个公开方法 `install` 。这个方法的第一个参数是 `Vue` 构造器，第二个参数是一个可选的选项对象：

```
MyPlugin.install = function (Vue, options) {
 // 1\. 添加全局方法或属性
 Vue.myGlobalMethod = ...
 // 2\. 添加全局资源
 Vue.directive('my-directive', {})
 // 3\. 添加实例方法
 Vue.prototype.$myMethod = ...
}
```

使用插件

通过 `Vue.use()` 全局方法使用插件：

```
// 调用 `MyPlugin.install(Vue)`
Vue.use(MyPlugin)
```

也可以传入一个选项对象：

```
Vue.use(MyPlugin, { someOption: true })
```

一些插件，如 `vue-router` ，如果 `Vue` 是全局变量则自动调用 `Vue.use()` 。不过在模块环境中应当始终显式调用 `Vue.use()` ：

```
// 通过 Browserify 或 Webpack 使用 CommonJS 兼容模块
var Vue = require('vue')
var VueRouter = require('vue-router')

// 不要忘了调用此方法
Vue.use(VueRouter)
```

已有插件 & 工具

- vue-router：Vue.js 官方路由。与 Vue.js 内核深度整合，让构建单页应用易如反掌。
- vue-resource：通过 XMLHttpRequest 或 JSONP 发起请求并处理响应。
- vue-async-data：异步加载数据插件。
- vue-validator：表单验证插件。
- vue-devtools：Chrome 开发者工具扩展，用于调试 Vue.js 应用。
- vue-touch：使用 Hammer.js 添加触摸手势指令（已过时）。
- vue-element：使用 Vue.js 注册自定义元素。
- 用户贡献的工具

构建大型应用

Vue.js 的设计思想是专注与灵活——它只是一个界面库，不强制使用哪个架构。它能很好地与已有项目整合，不过对于经验欠缺的开发者，从头开始构建大型应用可能是一个挑战。

Vue.js 生态系统提供了一系列的工具与库，用于构建单页应用。但是它们只是推荐而已。

模块化

对于大型项目，为了更好地管理代码使用模块构建系统非常必要。推荐代码使用 CommonJS 或 ES6 模块，然后使用 Webpack 或 Browserify 打包。

Webpack 和 Browserify 不只是模块打包器。两者都提供了源码转换 API，通过它可以用其它预处理器转换源码。例如，借助 babel-loader 或 babelify 代码可以使用 ES2015/2016 语法。

如果你之前没有用过它们，我强烈推荐你阅读一些教程，了解模块打包器，然后使用最新的 ECMAScript 特性写 JavaScript。

单文件组件

在典型的 Vue.js 项目中，我们会把界面拆分为多个小组件，每个组件在同一地方封装它的 CSS 样式，模板和 JavaScript 定义，这么做比较好。如上所述，使用 Webpack 或 Browserify 以及合适的源码转换器，我们可以这样写组件：



如果你喜欢预处理器，甚至可以这么做：



你可以使用 Webpack + vue-loader 或 Browserify + vueify 构建这些单文件 Vue 组件。推荐使用 Webpack，因为它的加载器 API 提供更好的文件依赖追踪/缓存以及一些 Browserify 没有的转换功能。

在 GitHub 上有一些构建示例：

- Webpack + vue-loader

- Browserify + vueify

路由

对于单页应用，推荐使用官方库 vue-router。详细请查看它的文档。

如果你只需要非常简单的路由逻辑，可以这么做，监听 `hashchange` 事件并使用动态组件：

示例：

```
<div id="app">
 <component :is="currentView"></component>
</div>
```

```
Vue.component('home', { /* ... */ })
Vue.component('page1', { /* ... */ })
var app = new Vue({
 el: '#app',
 data: {
 currentView: 'home'
 }
})
// 在路由处理器中切换页面
app.currentView = 'page1'
```

利用这种机制也可以非常容易地配合其它路由库，如 Page.js 或 Director。

与服务器通信

Vue 实例的原始数据 `$data` 能直接用 `JSON.stringify()` 序列化。社区贡献了一个插件 vue-resource，提供一种容易的方式与 RESTful APIs 配合。也可以使用任何自己喜欢的 Ajax 库，如 `$.ajax` 或 SuperAgent。Vue.js 也能很好地与无后端服务配合，如 Firebase 和 Parse。

状态管理

在大型应用中，状态管理常常变得复杂，因为状态分散在许多组件内。常常忽略 Vue.js 应用的来源是原生的数据对象—— Vue 实例代理访问它。因此，如果一个状态要被多个实例共享，应避免复制它：

```
var sourceOfTruth = {}

var vmA = new Vue({
 data: sourceOfTruth
})

var vmB = new Vue({
 data: sourceOfTruth
})
```

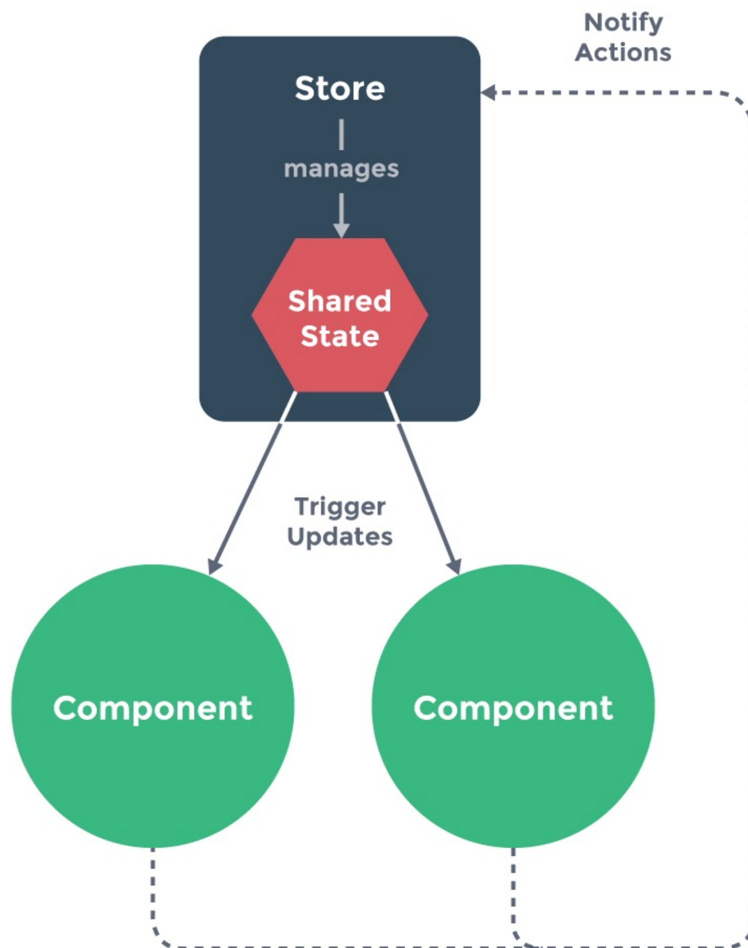
现在每当 `sourceOfTruth` 被修改后， `vmA` 与 `vmB` 将自动更新它们的视图。扩展这个思路，我们可以实现 **store** 模式：

```
var store = {
 state: {
 message: 'Hello!'
 },
 actionA: function () {
 this.state.message = 'action A triggered'
 },
 actionB: function () {
 this.state.message = 'action B triggered'
 }
}

var vmA = new Vue({
 data: {
 privateState: {},
 sharedState: store.state
 }
})

var vmB = new Vue({
 data: {
 privateState: {},
 sharedState: store.state
 }
})
```

我们把所有的 action 放在 store 内，action 修改 store 的状态。集中管理状态更易于理解状态将怎样变化。组件仍然可以拥有和管理它的私有状态。



有一点要注意，不要在 action 中替换原始的状态对象——为了观察到变化，组件和 store 需要共享这个对象。

如果我们约定，组件不可以直接修改 store 的状态，而应当派发事件，通知 store 执行 action，那么我们基本上实现了 Flux 架构。此约定的好处是，我们能记录 store 所有的状态变化，并且在此之上实现高级的调试帮助函数，如修改日志，快照，历史回滚等。

Flux 架构常用于 React 应用中。借助于响应系统，Flux 的核心思想能非常容易地用 Vue.js 实现。注意我们这里的演示只是为了介绍概念，简单的情况完全不需要这么做，应根据应用的需求调整这个模式。

单元测试

任何支持模块构建系统的单元测试工具都可以。推荐使用 Karma。它有许多插件，支持 Webpack 和 Browserify。用法见它们的文档。

代码测试的最佳实践是导出组件模块的选项/函数。例如：

```
// my-component.js
module.exports = {
 template: '<span>{{msg}}</span>',
 data: function () {
 return {
 msg: 'hello!'
 }
 }
 created: function () {
 console.log('my-component created!')
 }
}
```

在入口模块中使用这个模块：

```
// main.js
var Vue = require('vue')
var app = new Vue({
 el: '#app',
 data: { /* ... */ },
 components: {
 'my-component': require('./my-component')
 }
})
```

测试这个模块：

```
// Jasmine 2.0 测试
describe('my-component', function () {
 // require source module
 var myComponent = require('../src/my-component')
 it('should have a created hook', function () {
 expect(typeof myComponent.created).toBe('function')
 })
 it('should set correct default data', function () {
 expect(typeof myComponent.data).toBe('function')
 var defaultData = myComponent.data()
 expect(defaultData.msg).toBe('hello!')
 })
})
```

因为 Vue.js 指令是异步更新，如果想在修改数据之后修改 DOM，应当在 `Vue.nextTick` 的回调中操作。

生产发布

为了更小的文件体积，Vue.js 的压缩版本删除所有的警告，但是在使用 Browserify 或 Webpack 等工具构建 Vue.js 应用时，压缩需要一些配置。

Webpack

使用插件 DefinePlugin 将当前环境指定为生产环境，警告将在 UglifyJS 压缩代码过程中被删除。配置示例：

```
var webpack = require('webpack')

module.exports = {
 // ...
 plugins: [
 // ...
 new webpack.DefinePlugin({
 'process.env': {
 NODE_ENV: '"production"'
 }
 }),
 new webpack.optimize.UglifyJsPlugin({
 compress: {
 warnings: false
 }
 })
 ]
}
```

Browserify

将 NODE_ENV 设置为 “production”，然后运行打包命令。Vue 会自动应用 envify 并让警告块不能运行。例如：

```
NODE_ENV=production browserify -e main.js &#124; uglifyjs -c -m > b
```

应用示例

Vue.js Hackernews Clone 这个应用示例使用 Webpack + vue-loader 组织代码，使用 vue-router 作为路由器，HackerNews 官方的 Firebase API 作为后端。它当然不是大应用，但是它综合演示了本页讨论的概念。

对比其它框架

Angular

选择 Vue 而不选择 Angular，有下面几个原因，当然不是对每个人都适合：

- 在 API 与设计两方面上 Vue.js 都比 Angular 简单得多，因此你可以快速地掌握它的全部特性并投入开发。

- Vue.js 是一个更加灵活开放的解决方案。它允许你以希望的方式组织应用程序，而不是任何时候都必须遵循 Angular 制定的规则。它仅仅是一个视图层，所以你可以将它嵌入一个现有页面而不一定要做成一个庞大的单页应用。在配合其他库方面它给了你更大的的空间，但相应，你也需要做更多的架构决策。例如，Vue.js 核心默认不包含路由和 Ajax 功能，并且通常假定你在应用中使用了一个模块构建系统。这可能是最重要的区别。

- Angular 使用双向绑定，Vue 也支持双向绑定，不过默认为单向绑定，数据从父组件单向传给子组件。在大型应用中使用单向绑定让数据流易于理解。

- 在 Vue.js 中指令和组件分得更清晰。指令只封装 DOM 操作，而组件代表一个自给自足的独立单元 —— 有自己的视图和数据逻辑。在 Angular 中两者有不少相混的地方。

- Vue.js 有更好的性能，并且非常非常容易优化，因为它不使用脏检查。Angular，当 watcher 越来越多时会变得越来越慢，因为作用域内的每一次变化，所有 watcher 都要重新计算。并且，如果一些 watcher 触发另一个更新，脏检查循环（digest cycle）可能要运行多次。 Angular 用户常常要使用深奥的技术，以解决脏检查循环的问题。有时没有简单的办法来优化有大量 watcher 的作用域。Vue.js 则根本没有这个问题，因为它使用基于依赖追踪的观察系统并且异步列队更新，所有的数据变化都是独立地触发，除非它们之间有明确的依赖关系。唯一需要做的优化是在 `v-for` 上使用 `track-by` 。

有意思的是，Angular 2 和 Vue 用相似的设计解决了一些 Angular 1 中存在的问题。

React

React.js 和 Vue.js 确实有一些相似 —— 它们都提供数据驱动、可组合搭建的视图组件。当然它们也有许多不同。

首先，内部实现本质上不同。React 的渲染建立在 Virtual DOM 上——一种在内存中描述 DOM 树状态的数据结构。当状态发生变化时，React 重新渲染 Virtual DOM，比较计算之后给真实 DOM 打补丁。

Virtual DOM 提供了一个函数式的方法描述视图，这真的很棒。因为它不使用数据观察机制，每次更新都会重新渲染整个应用，因此从定义上保证了视图与数据的同步。它也开辟了 JavaScript 同构应用的可能性。

Vue.js 不使用 Virtual DOM 而是使用真实 DOM 作为模板，数据绑定到真实节点。Vue.js 的应用环境必须提供 DOM。但是，相对于常见的误解——Virtual DOM 让 React 比其它的都快， Vue.js 实际上性能比 React 好，而且几乎不用手工优化。而 React，为了最优化的渲染需要处处实现 `shouldComponentUpdate` 或使用不可变数据结构。

在 API 方面，React（或 JSX）的一个问题是，渲染函数常常包含大量的逻辑，最终看着更像是程序片断（实际上就是）而不是界面的视觉呈现。对于部分开发者来说，他们可能觉得这是个优点，但对那些像我一样兼顾设计和开发的人来说，模板能让我们更好地在视觉上思考设计和 CSS。JSX 和 JavaScript 逻辑的混合干扰了我将代码映射到设计的思维过程。相反，Vue.js 通过在模板中加入一个轻量级的 DSL (指令系统)，换来一个依旧直观的模板，且能将逻辑封装进指令和过滤器中。

React 的另一个问题是：由于 DOM 更新完全交给 Virtual DOM 管理，当想要自己控制 DOM 时就有点棘手了（虽然理论上可以做到，但是这样做就本质上违背了 React 的设计思想）。如果应用需要特别的自定义 DOM 操作，特别是复杂时间控制的动画，这个限制就很讨厌。在这方面，Vue.js 更灵活，有许多用 Vue.js 制作的 FWA/Awwwards 获奖站点。

再多说几句：

- React 团队雄心勃勃，计划让 React 成为通用平台的 UI 开发工具，而 Vue 专注于为 Web 提供实用的解决方案。
- React，由于它的函数式特质，可以很好地使用函数式编程模式。但是对于初级开发者和初学者这也导致较大的学习难度。Vue 更易学习并能快速投入开发。
- 对于大型应用，React 社区已经创造了大量的状态管理方案，例如 Flux/Redux。Vue 本身不解决这个问题（React 内核也是），但是可以轻松地修改状态管理模式，实现一个类似的架构。我已经看到有用户以 Vue 使用 Redux。Optimizely 的工程师也以 Vue 使用 NuclearJS (他们的 Flux 实现)。
- React 的开发趋势是将所有东西都放在 JavaScript 中，包括 CSS。已经有许多 CSS-in-JS 方案，但是所有的方案多多少少都有它的问题。而且更重要的是，这么做脱离了标准的 CSS 开发经验，并且很难和 CSS 社区的已有工作配合。Vue 的 单文件组件 在把 CSS 封装到组件模块的同时仍然允许你使用你喜欢的预处理器。

Ember

Ember 是一个全能框架。它提供大量的约定，一旦你熟悉了它们，开发会很高效。不过，这也意味着学习曲线较高，而且不灵活。在框架和库（加上一系列松散耦合的工具）之间权衡选择。后者更自由，但是也要求你做更多的架构决定。

也就是说，最好比较 Vue.js 内核和 Ember 的模板与数据模型层：

- Vue 在普通 JavaScript 对象上建立响应，提供自动化的计算属性。在 Ember 中需要将所有东西放在 Ember 对象内，并且手工为计算属性声明依赖。
- Vue 的模板语法可以用全功能的 JavaScript 表达式，而 Handlebars 的语法和帮助函数语法相比之下非常受限。
- 在性能上，Vue 甩开 Ember 几条街，即使是 Ember 2.0 最新的 Glimmer 引擎。Vue 自动批量更新，在性能比较关键时 Ember 要手工管理循环。

Polymer

Polymer 是另一个由 Google 支持的项目，实际上也是 Vue.js 的灵感来源之一。Vue.js 的组件可以类比为 Polymer 中的自定义元素，它们提供类似的开发体验。最大的不同在于，Polymer 依赖最新的 Web 组件特性，在不支持的浏览器中，需要加载笨重的 polyfill，性能也会受到影响。相对的，Vue.js 无需任何依赖，最低兼容到IE9。

另外，在 Polymer 1.0 中，为了性能开发团队严格限制了它的数据绑定系统。例如，Polymer 模板支持的表达式仅有逻辑逆运算和简单的方法调用。它的计算属性实现得也不是很灵活。

最后，当发布到生产环境时，Polymer 元素需要用专用工具 vulcanizer 打包。相比之下，单文件 Vue 组件能与 Webpack 无缝整合，因而你可以轻松在组件中使用 ES6 及任意 CSS 预处理器。

Riot

Riot 2.0 提供类似的基于组件的开发模式（Riot 称之为“标签”），API 小而美。我认为 Riot 与 Vue 在设计思路上有许多相同点。不过，尽管比 Riot 重一点，Vue 提供了一些显著优处：

- 真实的条件渲染，Riot 渲染所有的分支，然后简单地显示/隐藏它们。
- 一个强大得多的路由器，Riot 的路由 API 过于简陋。
- 更成熟的工具链支持，见 Webpack + vue-loader。
- 过渡效果系统，Riot 没有。
- 更佳的性能。Riot 实际上使用脏检查而不是 Virtual DOM，因而遭受跟 Angular 一样的性能问题。

API

全局配置

`Vue.config` 是一个对象，包含 Vue 的全局配置。可以在启动应用之前修改下面属性：

debug

- 类型： `Boolean`
- 默认值： `false`
- 用法：

  ```
  Vue.config.debug = true
  ```

  在调试模式中，Vue 会：

  1. 为所有的警告打印栈追踪。
  2. 把所有的锚节点以注释节点显示在 DOM 中，更易于检查渲染结果的结构。

只有开发版本可以使用调试模式。

delimiters

- 类型： `Array<String>`
- 默认值： `["{{", "}}"]`
- 用法：

  ```
  // ES6 模板字符串
  Vue.config.delimiters = ['${', '}']
  ```

  修改文本插值的定界符。

unsafeDelimiters

- 类型： `Array<String>`

- 默认值： `["{{{", "}}}"]`
- 用法：

  ```
  // make it look more dangerous
  Vue.config.unsafeDelimiters = ['{!!', '!!}']
  ```

  修改原生 HTML 插值的定界符。

silent

- 类型： `Boolean`
- 默认值： `false`
- 用法：

  ```
  Vue.config.silent = true
  ```

  取消 Vue.js 所有的日志与警告。

async

- 类型： `Boolean`
- 默认值： `true`
- 用法：

  ```
  Vue.config.async = false
  ```

  如果关闭了异步模式，Vue 在检测到数据变化时同步更新 DOM。在有些情况下这有助于调试，但是也可能导致性能下降，并且影响 watcher 回调的调用顺序。 **`async: false`** 不推荐用在生产环境中。

convertAllProperties

- 类型： `Boolean`
- 默认值： `false`
- 用法：

  ```
  Vue.config.convertAllProperties = true
  ```

1.0.8 添加。开启这个选项后，Vue 可以转换和观察由 `Object.defineProperty` 定义的 getters/setter。默认关闭，因为会付出一些性能代价，并且不是常用功能。

全局 API

Vue.extend(options)

- 参数：
  - `{Object} options`
- 用法：

  创建基础 Vue 构造器的“子类”。参数是一个对象，包含组件选项。

  这里要注意的特例是 `el` 和 `data` 选项—— 在 `Vue.extend()` 中它们必须是函数。

```
<div id="mount-point"></div>
```

```
// 创建可复用的构造器
var Profile = Vue.extend({
 template: '<p>{{firstName}} {{lastName}} aka {{alias}}</p>'
})
// 创建一个 Profile 实例
var profile = new Profile({
 data: {
 firstName: 'Walter',
 lastName: 'White',
 alias: 'Heisenberg'
 }
})
// 挂载到元素上
profile.$mount('#mount-point')
```

  结果：

```
<p>Walter White aka Heisenberg</p>
```

- 另见：组件

Vue.nextTick(callback)

- 参数：

  - `{Functon} callback`
- 用法：

  延迟回调在下次 DOM 更新循环之后执行。在修改数据之后立即使用这个方法，等待 DOM 更新。

  ```
  // 修改数据
  vm.msg = 'Hello'
  // DOM 没有更新
  Vue.nextTick(function () {
   // DOM 更新了
  })
  ```

- 另见：异步更新队列

Vue.set(object, key, value)

- 参数：

  - `{Object} object`
  - `{String} key`
  - `{*} value`
- 返回值：设置的值

- 用法：

  设置对象的属性。如果对象是响应的，将触发视图更新。这个方法主要用于解决 不能检测到属性添加的限制。

- 另见：深入响应

Vue.delete(object, key)

- 参数：

  - `{Object} object`
  - `{String} key`
- 用法：

  删除对象的属性。如果对象是响应的，将触发视图更新。这个方法主要用于解决 不能检测到属性删除的限制。

- 另见：深入响应

Vue.directive(id, [definition])

- 参数：

  - `{String} id`
  - `{Function | Object} [definition]`
- 用法：

  注册或获取全局指令。

```
// 注册
Vue.directive('my-directive', {
  bind: function () {},
  update: function () {},
  unbind: function () {}
})

// 注册，传入一个函数
Vue.directive('my-directive', function () {
  // this will be called as `update`
})

// getter，返回已注册的指令
var myDirective = Vue.directive('my-directive')
```

- 另见：自定义指令

Vue.elementDirective(id, [definition])

- 参数：
  - `{String} id`
  - `{Object} [definition]`
- 用法：

  注册或获取全局的元素指令。

```
// 注册
Vue.elementDirective('my-element', {
  bind: function () {},
  // 没有使用 `update`
  unbind: function () {}
})

// getter，返回已注册的元素指令
var myDirective = Vue.elementDirective('my-element')
```

- 另见：元素指令

Vue.filter(id, [definition])

- 参数：

  - `{String} id`
  - `{Function | Object} [definition]`
- 用法：

  注册或获取全局过滤器。

```
// 注册
Vue.filter('my-filter', function (value) {
  // 返回处理后的值
})

// 双向过滤器
Vue.filter('my-filter', {
  read: function () {},
  write: function () {}
})

// getter，返回已注册的指令
var myFilter = Vue.filter('my-filter')
```

- 另见：自定义过滤器

Vue.component(id, [definition])

- 参数：
  - `{String} id`
  - `{Function | Object} [definition]`
- 用法：

  注册或获取全局组件。

```
// 注册，传入一个扩展的构造器
Vue.component('my-component', Vue.extend({ /* ... */}))

// 注册，传入一个选项对象（自动调用 Vue.extend）
Vue.component('my-component', { /* ... */ })

// 获取注册的组件（始终返回构造器）
var MyComponent = Vue.component('my-component')
```

- 另见：组件.

Vue.transition(id, [hooks])

- 参数：

    - `{String} id`
    - `{Object} [hooks]`
- 用法：

  注册或获取全局的过渡钩子对象。

```
// 注册
Vue.transition('fade', {
  enter: function () {},
  leave: function () {}
})

// 获取注册的钩子
var fadeTransition = Vue.transition('fade')
```

- 另见：过渡.

Vue.partial(id, [partial])

- 参数：
    - `{String} id`
    - `{String} [partial]`
- 用法：

  注册或获取全局的 partial。

```
// 注册
Vue.partial('my-partial', '<div>Hi</div>')

// 获取注册的 partial
var myPartial = Vue.partial('my-partial')
```

- 另见：特殊元素 - 。

Vue.use(plugin, [options])

- 参数：
    - `{Object | Function} plugin`
    - `{Object} [options]`
- 用法：

  安装 Vue.js 插件。如果插件是一个对象，必须有一个 `install` 方法。如果它是一个函数，它会被作为安装方法。安装方法以 Vue 为参数。

- 另见：插件.

Vue.mixin(mixin)

- 参数：
  - `{Object} mixin`
- 用法：

  全局应用一个混合，将影响所有 Vue 实例。插件作者可以用它向组件注入自定义逻辑。不推荐用在应用代码中。

- 另见：全局混合

选项 / 数据

data

- 类型：`Object | Function`
- 限制：在 `Vue.extend()` 中只能是函数。
- 详细：

  Vue 实例的数据对象。Vue.js 会递归地将它全部属性转为 getter/setter，从而让它能响应数据变化。这个对象必须是普通对象：原生对象，getter/setter 及原型属性会被忽略。不推荐观察复杂对象。

  在实例创建之后，可以用 `vm.$data` 访问原始数据对象。Vue 实例也代理了数据对象所有的属性。

  名字以 `_` 或 `$` 开始的属性不会被 Vue 实例代理，因为它们可能与 Vue 的内置属性与 API 方法冲突。用 `vm.$data._property` 访问它们。

  可以通过将 `vm.$data` 传入 `JSON.parse(JSON.stringify(...))` 得到原始数据对象。

- 示例：

```
var data = { a: 1 }

// 直接创建一个实例
var vm = new Vue({
 data: data
})
vm.a // -> 1
vm.$data === data // -> true

// 在 Vue.extend() 中必须是函数
var Component = Vue.extend({
 data: function () {
 return { a: 1 }
 }
})
```

- 另见：深入响应.

props

- 类型： `Array | Object`
- 详细：

  包含一些特性——期望使用的父组件数据的属性。可以是数组或对象。对象用于高级配置，如类型检查，自定义验证，默认值等。

- 示例：

```
// 简单语法
Vue.component('props-demo-simple', {
 props: ['size', 'myMessage']
})

// 对象语法，指定验证要求
Vue.component('props-demo-advanced', {
 props: {
 // 只检测类型
 size: Number,
 // 检测类型 + 其它验证
 name: {
 type: String,
 required: true
 }
 }
})
```

- 另见： Props

computed

- 类型： `Object`
- 详细：

  实例计算属性。getter 和 setter 的 `this` 自动地绑定到实例。

- 示例：

```
var vm = new Vue({
 data: { a: 1 },
 computed: {
 // 仅读取，值只须为函数
 aDouble: function () {
 return this.a * 2
 },
 // 读取和设置
 aPlus: {
 get: function () {
 return this.a + 1
 },
 set: function (v) {
 this.a = v - 1
 }
 }
 }
})
vm.aPlus   // -> 2
vm.aPlus = 3
vm.a       // -> 2
vm.aDouble // -> 4
```

- 另见：
  - 计算属性
  - 深入响应：计算属性

methods

- 类型： `Object`
- 详细：

  实例方法。实例可以直接访问这些方法，也可以用在指令表达式内。方法的 `this` 自动绑定到实例。

- 示例：

```
var vm = new Vue({
 data: { a: 1 },
 methods: {
 plus: function () {
 this.a++
 }
 }
})
vm.plus()
vm.a // 2
```

- 另见： 方法与事件处理器

watch

- 类型： `Object`
- 详细：

  一个对象，键是观察表达式，值是对应回调。值也可以是方法名，或者是对象，包含选项。在实例化时为每个键调用 `$watch()` 。

- 示例：

```
var vm = new Vue({
 data: {
 a: 1
 },
 watch: {
 'a': function (val, oldVal) {
 console.log('new: %s, old: %s', val, oldVal)
 },
 // 方法名
 'b': 'someMethod',
 // 深度 watcher
 'c': {
 handler: function (val, oldVal) { /* ... */ },
 deep: true
 }
 }
})
vm.a = 2 // -> new: 2, old: 1
```

- 另见： 实例方法 - vm.$watch

选项 / DOM

el

- 类型： `String | HTMLElement | Function`
- 限制： `Vue.extend()` 内只能是函数
- 详细：

  为实例提供挂载元素。值可以是 CSS 选择符，或实际 HTML 元素，或返回 HTML 元素的函数。注意元素只用作挂载点。如果提供了模板则元素被替换，除非 `replace` 为 false。元素可以用 `vm.$el` 访问。

  用在 `Vue.extend` 中必须是函数值，这样所有实例不会共享元素。

  如果在初始化时指定了这个选项，实例将立即进入编译过程。否则，需要调用 `vm.$mount()`，手动开始编译。

- 另见： 生命周期图示

template

- 类型： `String`
- 详细：

  实例模板。模板默认替换挂载元素。如果 `replace` 选项为 `false`，模板将插入挂载元素内。两种情况下，挂载元素的内容都将被忽略，除非模板有内容分发 slot。

  如果值以 `#` 开始，则它用作选项符，将使用匹配元素的 innerHTML 作为模板。常用的技巧是用 `<script type="x-template">` 包含模板。

  注意在一些情况下，例如如模板包含多个顶级元素，或只包含普通文本，实例将变成一个片断实例，管理多个节点而不是一个节点。片断实例的挂载元素上的非流程控制指令被忽略。

- 另见：
  - 生命周期图示
  - 使用 slot 分发内容
  - 片断实例

replace

- 类型： `Boolean`
- 默认值： `true`
- 限制： 只能与 **template** 选项一起用
- 详细：

决定是否替换模板的挂载元素。如果设为 `false` 模板将覆盖元素的内容，不会替换元素本身。

- 示例：

```
<div id="replace"></div>
```

```
new Vue({
 el: '#replace',
 template: '<p>replaced</p>'
})
```

结果：

```
<p>replaced</p>
```

`replace` 设为 `false` ：

```
<div id="insert"></div>
```

```
new Vue({
 el: '#insert',
 replace: false,
 template: '<p>inserted</p>'
})
```

结果：

```
<div id="insert">
 <p>inserted</p>
</div>
```

选项 / 生命周期钩子

created

- 类型： `Function`
- 详细：

在实例创建之后同步调用。此时实例已经结束解析选项，这意味着已建立：数据绑定，计算属性，方法，watcher/事件回调。但是还没有开始 DOM 编译，`$el` 还不存在。

- 另见：生命周期图示

beforeCompile

- 类型：`Function`
- 详细：

  在编译开始前调用。

- 另见：生命周期图示

compiled

- 类型：`Function`
- 详细：

  在编译结束后调用。此时所有的指令已生效，因而数据的变化将触发 DOM 更新。但是不担保 `$el` 已插入文档。

- 另见：生命周期图示

ready

- 类型：`Function`
- 详细：

  在编译结束和 **`$el`** 第一次插入文档之后调用，如在第一次 `attached` 钩子之后调用。注意必须是由 Vue 插入（如 `vm.$appendTo()` 等方法或指令更新）才触发 `ready` 钩子。

- 另见：生命周期图示

attached

- 类型：`Function`
- 详细：

  在 `vm.$el` 插入 DOM 时调用。必须是由指令或实例方法（如 `$appendTo()`）插入，直接操作 `vm.$el` 不会 触发这个钩子。

detached

- 类型： `Function`
- 详细：

  在 `vm.$el` 从 DOM 中删除时调用。必须是由指令或实例方法删除，直接操作 `vm.$el` 不会 触发这个钩子。

beforeDestroy

- 类型： `Function`
- 详细：

  在开始销毁实例时调用。此时实例仍然有功能。

- 另见： 生命周期图示

destroyed

- 类型： `Function`
- 详细：

  在实例被销毁之后调用。此时所有的绑定和实例的指令已经解绑，所有的子实例也已经被销毁。

  如果有离开过渡， `destroyed` 钩子在过渡完成之后调用。

- 另见： 生命周期图示

选项 / 资源

directives

- 类型： `Object`
- 详细：

  一个对象，包含指令。

- 另见：
  - 自定义指令
  - 资源命名约定

elementDirectives

- 类型： `Object`

- 详细：

  一个对象，包含元素指令。

- 另见：

  - 元素指令
  - 资源命名约定

filters

- 类型： `Object`

- 详细：

  一个对象，包含过滤器。

- 另见：

  - 自定义过滤器
  - 资源命名约定

components

- 类型： `Object`

- 详细：

  一个对象，包含组件。

- 另见：

  - 组件

transitions

- 类型： `Object`

- 详细：

  一个对象，包含过渡。

- 另见：

  - 过渡

partials

- 类型： `Object`

- 详细：

一个对象，包含 partial。

- 另见：
  - 特殊元素 - partial

选项 / 杂项

parent

- 类型：`Vue 实例`
- 详细：

  指定实例的父实例，在两者之间建立父子关系。子实例可以用 `this.$parent` 访问父实例，子实例被推入父实例的 `$children` 数组中。

- 另见：父子组件通信

events

- 类型：`Object`
- 详细：

  一个对象，键是监听的事件，值是相应的回调。注意这些事件是 Vue 事件而不是 DOM 事件。值也可以是方法的名字。在实例化的过程中，Vue 实例会调用对象的每个键。

- 示例：

```
var vm = new Vue({
 events: {
 'hook:created': function () {
 console.log('created!')
 },
 greeting: function (msg) {
 console.log(msg)
 },
 // 也可以是方法的名字
 bye: 'sayGoodbye'
 },
 methods: {
 sayGoodbye: function () {
 console.log('goodbye!')
 }
 }
}) // -> created!
vm.$emit('greeting', 'hi!') // -> hi!
vm.$emit('bye')             // -> goodbye!
```

- 另见：
  - 实例方法 - 事件
  - 父子组件通信

mixins

- 类型： `Array`
- 详细：

  一个数组，包含混合对象。这些混合对象可以像普通实例对象一样包含实例选项，它们将合并成一个最终选项对象，合并策略同 `Vue.extend()` 。比如，如果混合对象包含一个 `created` 钩子，组件自身也包含一个，两个钩子函数都会被调用。

  混合后的钩子按它们出现顺序调用，并且是在调用组件自己的钩子之前调用。

- 示例：

```
var mixin = {
 created: function () { console.log(1) }
}
var vm = new Vue({
 created: function () { console.log(2) },
 mixins: [mixin]
})
// -> 1
// -> 2
```

- 另见：混合

name

- 类型：`String`
- 限制：只能用在 `Vue.extend()` 中。
- 详细：

  允许组件在它的模板内递归地调用它自己。注意如果组件是由 `Vue.component()` 全局注册，全局 ID 自动作为它的名字。

  指定 `name` 选项的另一个好处是方便检查。当在控制台检查组件时，默认的构造器名字是 `VueComponent`，不大有用。在向 `Vue.extend()` 传入 `name` 选项后，可以知道正在检查哪个组件。值会被转换为驼峰形式，并用作组件构造器的名字。

- 示例：

```
var Ctor = Vue.extend({
 name: 'stack-overflow',
 template:
 '<div>' +
 // 递归地调用自己
 '<stack-overflow></stack-overflow>' +
 '</div>'
})

// 将导致错误：Maximum call stack size exceeded
// 不过我们假定没问题...
var vm = new Ctor()

console.log(vm) // -> StackOverflow {$el: null, ...}
```

实例属性

vm.$data

- 类型：`Object`
- 详细：

  Vue 实例观察的数据对象。可以用一个新的对象替换。实例代理了它的数据对象的属性。

vm.$el

- 类型： `HTMLElement`
- 只读
- 详细：

  Vue 实例的挂载元素。注意对于片段实例， `vm.$el` 返回一个锚节点，指示片断的开始位置。

vm.$options

- 类型： `Object`
- 只读
- 详细：

  当前实例的初始化选项。在选项中包含自定义属性时有用处：

```
new Vue({
 customOption: 'foo',
 created: function () {
 console.log(this.$options.customOption) // -> 'foo'
 }
})
```

vm.$parent

- 类型： `Vue 实例`
- 只读
- 详细：

  父实例，如果当前实例有的话。

vm.$root

- 类型： `Vue 实例`
- 只读
- 详细：

  当前组件树的根 Vue 实例。如果当前实例没有父实例，值将是它自身。

vm.$children

- 类型： `Array<Vue instance>`

- 只读
- 详细：

  当前实例的直接子组件。

vm.$refs

- 类型： `Object`
- 只读
- 详细：

  一个对象，包含注册有 `v-ref` 的子组件。

- 另见：
  - 子组件索引
  - v-ref

vm.$els

- 类型： `Object`
- 只读
- 详细：

  一个对象，包含注册有 `v-el` 的 DOM 元素。

- 另见： v-el。

实例方法 / 数据

vm.$watch(expOrFn, callback, [options])

- 参数：
  - `{String|Function} expOrFn`
  - `{Function} callback`
  - `{Object} [options]`
    - `{Boolean} deep`
    - `{Boolean} immediate`
- 返回值： `{Function} unwatch`
- 用法：

观察 Vue 实例的一个表达式或计算函数。回调的参数为新值和旧值。表达式可以是某个键路径或任意合法绑定表达式。

注意：在修改（不是替换）对象或数组时，旧值将与新值相同，因为它们索引同一个对象/数组。Vue 不会保留修改之前值的副本。

- 示例：

```
// 键路径
vm.$watch('a.b.c', function (newVal, oldVal) {
 // 做点什么
})

// 表达式
vm.$watch('a + b', function (newVal, oldVal) {
 // 做点什么
})

// 函数
vm.$watch(
 function () {
 return this.a + this.b
 },
 function (newVal, oldVal) {
 // 做点什么
 }
)
```

`vm.$watch` 返回一个取消观察函数，用来停止触发回调：

```
var unwatch = vm.$watch('a', cb)
// 之后取消观察
unwatch()
```

- **Option: deep**

为了发现对象内部值的变化，可以在选项参数中指定 `deep: true` 。注意监听数组的变动不需要这么做。

```
vm.$watch('someObject', callback, {
 deep: true
})
vm.someObject.nestedValue = 123
// 触发回调
```

- **Option: immediate**

在选项参数中指定 `immediate: true` 将立即以表达式的当前值触发回调：

```
vm.$watch('a', callback, {
 immediate: true
})
// 立即以 `a` 的当前值触发回调
```

vm.$get(expression)

- 参数：
  - `{String} expression`
- 用法：

  从 Vue 实例获取指定表达式的值。如果表达式抛出错误，则取消错误并返回 `undefined` 。

- 示例：

```
var vm = new Vue({
 data: {
 a: {
 b: 1
 }
 }
})
vm.$get('a.b') // -> 1
vm.$get('a.b + 1') // -> 2
```

vm.$set(keypath, value)

- 参数：
  - `{String} keypath`
  - `{*} value`
- 用法：

  设置 Vue 实例的属性值。多数情况下应当使用普通对象语法，如 `vm.a.b = 123` 。这个方法只用于下面情况：

  1. 使用 keypath 动态地设置属性。
  2. 设置不存在的属性。

  如果 keypath 不存在，将递归地创建并建立追踪。如果用它创建一个顶级属性，实例将被强制进入“ digest 循环”，在此过程中重新计算所有的 watcher。

- 示例：

```
var vm = new Vue({
 data: {
 a: {
 b: 1
 }
 }
})

// keypath 存在
vm.$set('a.b', 2)
vm.a.b // -> 2

// keypath 不存在
vm.$set('c', 3)
vm.c // ->
```

- 另见：深入响应

vm.$delete(key)

- 参数：
    - `{String} key`
- 用法：

  删除 Vue 实例（以及它的 `$data`）上的顶级属性。强制 digest 循环，不推荐使用。

vm.$eval(expression)

- 参数：
    - `{String} expression`
- 用法：

  计算当前实例上的合法的绑定表达式。表达式也可以包含过滤器。

- 示例：

```
// 假定 vm.msg = 'hello'
vm.$eval('msg &#124; uppercase') // -> 'HELLO'
```

vm.$interpolate(templateString)

- 参数：
    - `{String} templateString`

- 用法：

  计算模板，模板包含 Mustache 标签。注意这个方法只是简单计算插值，模板内的指令将被忽略。

- 示例：

```
// 假定 vm.msg = 'hello'
vm.$interpolate('{{msg}} world!') // -> 'hello world!'
```

vm.$log([keypath])

- 参数：
  - `{String} [keypath]`
- 用法：

  打印当前实例的数据，比起一堆 getter/setter 要友好。keypath 可选。

```
vm.$log() // 打印整个 ViewModel 的数据
vm.$log('item') // 打印 vm.item
```

实例属性 / 事件

vm.$on(event, callback)

- 参数：
  - `{String} event`
  - `{Function} callback`
- 用法：

  监听当前实例上的自定义事件。事件可以由 `vm.$emit`，`vm.$dispatch` 或 `vm.$broadcast` 触发。传入这些方法的附加参数都会传入这个方法的回调。

- 示例：

```
vm.$on('test', function (msg) {
 console.log(msg)
})
vm.$emit('test', 'hi')
// -> "hi"
```

vm.$once(event, callback)

- 参数：
  - `{String} event`
  - `{Function} callback`
- 用法：

  监听一个自定义事件，但是只触发一次，在第一次触发之后删除监听器。

vm.$off([event, callback])

- 参数：
  - `{String} [event]`
  - `{Function} [callback]`
- 用法：

  删除事件监听器。

  - 如果没有参数，则删除所有的事件监听器；
  - 如果只提供了事件，则删除这个事件所有的监听器；
  - 如果同时提供了事件与回调，则只删除这个回调。

vm.$emit(event, [...args])

- 参数：
  - `{String} event`
  - `[...args]`

  触发当前实例上的事件。附加参数都会传给监听器回调。

vm.$dispatch(event, [...args])

- 参数：
  - `{String} event`
  - `[...args]`
- 用法：

  派发事件，首先在实例上触发它，然后沿着父链向上冒泡在触发一个监听器后停止，除非它返回 `true` 。附加参数都会传给监听器回调。

- 示例：

```
// 创建父链
var parent = new Vue()
var child1 = new Vue({ parent: parent })
var child2 = new Vue({ parent: child1 })

parent.$on('test', function () {
 console.log('parent notified')
})
child1.$on('test', function () {
 console.log('child1 notified')
})
child2.$on('test', function () {
 console.log('child2 notified')
})

child2.$dispatch('test')
// -> "child2 notified"
// -> "child1 notified"
// 没有通知 parent，因为 child1 的回调没有返回 true
```

- 另见： 父子组件通信

vm.$broadcast(event, [...args])

- 参数：
  - `{String} event`
  - `[...args]`
- 用法：

  广播事件，通知给当前实例的全部后代。因为后代有多个枝杈，事件将沿着各“路径”通知。每条路径上的通知在触发一个监听器后停止，除非它返回 `true` 。

- 示例：

```
var parent = new Vue()
// child1 和 child2 是兄弟
var child1 = new Vue({ parent: parent })
var child2 = new Vue({ parent: parent })
// child3 在 child2 内
var child3 = new Vue({ parent: child2 })

child1.$on('test', function () {
 console.log('child1 notified')
})
child2.$on('test', function () {
 console.log('child2 notified')
})
child3.$on('test', function () {
 console.log('child3 notified')
})

parent.$broadcast('test')
// -> "child1 notified"
// -> "child2 notified"
// 没有通知 child3，因为 child2 的回调没有返回 true
```

实例方法 / DOM

vm.$appendTo(elementOrSelector, [callback])

- 参数：
  - `{Element|String} elementOrSelector`
  - `{Function} [callback]`
- 返回值： `vm` ——实例自身
- 用法：

  将实例的 DOM 元素或片断插入目标元素内。第一个参数可以是一个元素或选择器字符串。如果有过渡则触发过渡。回调在过渡完成后执行，如果没有触发过渡则立即执行。

vm.$before(elementOrSelector, [callback])

- 参数：
  - `{Element|String} elementOrSelector`
  - `{Function} [callback]`
- 返回值： `vm` ——实例自身
- 用法：

将实例的 DOM 元素或片断插到目标元素的前面。第一个参数可以是一个元素或选择器字符串。如果有过渡则触发过渡。回调在过渡完成后执行，如果没有触发过渡则立即执行。

vm.$after(elementOrSelector, [callback])

- 参数：
  - `{Element|String} elementOrSelector`
  - `{Function} [callback]`
- 返回值： `vm` ——实例自身
- 用法：

  将实例的 DOM 元素或片断插到目标元素的后面。第一个参数可以是一个元素或选择器字符串。如果有过渡则触发过渡。回调在过渡完成后执行，如果没有触发过渡则立即执行。

vm.$remove([callback])

- 参数：
  - `{Function} [callback]`
- 返回值： `vm` ——实例自身
- 用法：

  从 DOM 中删除实例的 DOM 元素或片断。如果有过渡则触发过渡。回调在过渡完成后执行，如果没有触发过渡则立即执行。

vm.$nextTick(callback)

- 参数：
  - `{Function} [callback]`
- 用法：

  将回调延迟到下次 DOM 更新循环之后执行。在修改数据之后立即使用它，然后等待 DOM 更新。它跟全局方法 `Vue.nextTick` 一样，不同的是回调的 `this` 自动绑定到调用它的实例上。

- 示例：

```
new Vue({
 // ...
 methods: {
 // ...
 example: function () {
 // 修改数据
 this.message = 'changed'
 // DOM 还没有更新
 this.$nextTick(function () {
 // DOM 现在更新了
 // `this` 绑定到当前实例
 this.doSomethingElse()
 })
 }
 }
})
```

- 另见：
  - Vue.nextTick
  - 异步更新队列

实例方法 / 生命周期

vm.$mount([elementOrSelector])

- 参数：
  - `{Element|String} [elementOrSelector]`
- 返回值： `vm` ——实例自身
- 用法：

  如果 Vue 实例在实例化时没有收到 `el` 选项，则它处于“未挂载”状态，没有关联的 DOM 元素或片断。可以使用 `vm.$mount()` 手动地开始挂载/编译未挂载的实例。

  如果没有参数，模板将被创建为文档之外的的片断，需要手工用其它的 DOM 实例方法把它插入文档中。如果 `replace` 选项为 `false` ，则自动创建一个空 `<div>` ，作为包装元素。

  在已经挂载的实例上调用 `$mount()` 没有效果。这个方法返回实例自身，因而可以链式调用其它实例方法。

- 示例：

```
var MyComponent = Vue.extend({
  template: '<div>Hello!</div>'
})

// 创建并挂载到 #app (会替换 #app)
new MyComponent().$mount('#app')

// 同上
new MyComponent({ el: '#app' })

// 手动挂载
new MyComponent().$mount().$appendTo('#container')
```

- 另见：生命周期图示

vm.$destroy([remove])

- 参数：
  - `{Boolean} [remove] - default: false`
- 用法：

  完全销毁实例。清理它与其它实例的连接，解绑它的全部指令及事件监听器，如果 `remove` 参数是 `true`，则从 DOM 中删除它关联的 DOM 元素或片断。

  触发 `beforeDestroy` 和 `destroyed` 钩子。

- 另见：生命周期图示

指令

v-text

- 类型：`String`
- 详细：

  更新元素的 `textContent`。

  在内部，`{{ Mustache }}` 插值也被编译为 textNode 的一个 `v-text` 指令。这个指令需要一个包装元素，不过性能稍好并且避免 FOUC (Flash of Uncompiled Content)。

- 示例：

```
<span v-text="msg"></span>
<!-- same as -->
<span>{{msg}}</span>
```

v-html

- 类型： `String`
- 详细：

  更新元素的 `innerHTML` 。内容按普通 HTML 插入——数据绑定被忽略。如果想复用模板片断，应当使用 partials。

  在内部， `{{{ Mustache }}}` 插值也会被编译为锚节点上的一个 `v-html` 指令。这个指令需要一个包装元素，不过性能稍好并且避免 FOUC (Flash of Uncompiled Content) 。

  在网站上动态渲染任意 HTML 是非常危险的，因为容易导致 XSS 攻击。只在可信内容上使用 `v-html` ，永不用在用户提交的内容上。

- 示例：

```
<div v-html="html"></div>
<!-- 相同 -->
<div>{{{html}}}</div>
```

v-if

- 类型： `*`
- 用法：

  根据表达式的值的真假条件渲染元素。在切换时元素及它的数据绑定 / 组件被销毁并重建。如果元素是 `<template>` ，将提出它的内容作为条件块。

- 另见： 条件渲染

v-show

- 类型： `*`
- 用法：

  根据表达式的值的真假切换元素的 `display` CSS 属性，如果有过渡将触发它。

- 另见： 条件渲染 - v-show

v-else

- 不需要表达式
- 限制：前一兄弟元素必须有 `v-if` 或 `v-show` 。
- 用法：

  为 `v-if` 和 `v-show` 添加 “else 块”。

  ```
  <div v-if="Math.random() > 0.5">
   Sorry
  </div>
  <div v-else>
   Not sorry
  </div>
  ```

- 另见：条件渲染 - v-else

v-for

- 类型： `Array | Object | Number | String`
- **Param Attributes:**
  - `track-by`
  - `stagger`
  - `enter-stagger`
  - `leave-stagger`
- 用法：

  基于源数据将元素或模板块重复数次。表达式必须使用特定语法，为当前遍历的元素提供别名：

  ```
  <div v-for="item in items">
   {{ item.text }}
  </div>
  ```

  另外也可以为数组索引指定别名（如果值是对象可以为键指定别名）：

  ```
  <div v-for="(index, item) in items"></div>
  <div v-for="(key, val) in object"></div>
  ```

- 另见：列表渲染

v-on

- 缩写： `@`
- 类型： `Function | Inline Statement`
- 参数： `event (required)`
- 修饰符：
    - `.stop` - 调用 `event.stopPropagation()` 。
    - `.prevent` - 调用 `event.preventDefault()` 。
    - `.{keyCode | keyAlias}` - 只在指定按键上触发回调。
- 用法：

  绑定事件监听器。事件类型由参数指定。表达式可以是一个方法的名字或一个内联语句，如果没有修饰符也可以省略。

  用在普通元素上时，只能监听原生 **DOM** 事件。用在自定义元素组件上时，也可以监听子组件触发的自定义事件。

  在监听原生 DOM 事件时，方法以事件为唯一的参数。如果使用内联语句，语句可以访问一个 `$event` 属性： `v-on:click="handle('ok', $event)"` 。

  1.0.11+ 在监听自定义事件时，内联语句可以访问一个 `$arguments` 属性，它是一个数组，包含传给子组件的 `$emit` 回调的参数。

- 示例：

```
<!-- 方法处理器 -->
<button v-on:click="doThis"></button>

<!-- 内联语句 -->
<button v-on:click="doThat('hello', $event)"></button>

<!-- 缩写 -->
<button @click="doThis"></button>

<!-- 停止冒泡 -->
<button @click.stop="doThis"></button>

<!-- 阻止默认行为 -->
<button @click.prevent="doThis"></button>

<!-- 阻止默认行为，没有表达式 -->
<form @submit.prevent></form>

<!-- 串联修饰符 -->
<button @click.stop.prevent="doThis"></button>

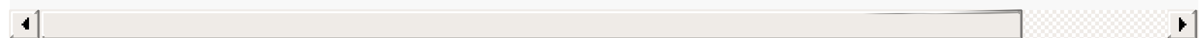
<!-- 键修饰符，键别名 -->
<input @keyup.enter="onEnter">

<!-- 键修饰符，键代码 -->
<input @keyup.13="onEnter">
```

在子组件上监听自定义事件（当子组件触发 “my-event” 时将调用事件处理器）：

```
<my-component @my-event="handleThis"></my-component>

<!-- 内联语句 -->
<my-component @my-event="handleThis(123, $arguments)"></my-comp
```



- 另见： 方法与事件处理器

v-bind

- 缩写： `:`
- 类型： `* (with argument) | Object (without argument)`
- 参数： `attrOrProp (optional)`
- 修饰符：
  - `.sync` - 双向绑定，只能用于 prop 绑定。

  - `.once` - 单次绑定，只能用于 prop 绑定。
- 用法：

  动态地绑定一个或多个 attribute，或一个组件 prop 到表达式。

  在绑定 `class` 或 `style` 时，支持其它类型的值，如数组或对象。

  在绑定 prop 时，prop 必须在子组件中声明。可以用修饰符指定不同的绑定类型。

  没有参数时，可以绑定到一个对象。注意此时 `class` 和 `style` 绑定不支持数组和对象。

- 示例：

```
<!-- 绑定 attribute -->
<img v-bind:src="imageSrc">

<!-- 缩写 -->
<img :src="imageSrc">

<!-- 绑定 class -->
<div :class="{ red: isRed }"></div>
<div :class="[classA, classB]"></div>

<!-- 绑定 style -->
<div :style="{ fontSize: size + 'px' }"></div>
<div :style="[styleObjectA, styleObjectB]"></div>

<!-- 绑定到一个对象 -->
<div v-bind="{ id: someProp, 'other-attr': otherProp }"></div>

<!-- prop 绑定，"prop" 必须在 my-component 组件内声明 -->
<my-component :prop="someThing"></my-component>

<!-- 双向 prop 绑定 -->
<my-component :prop.sync="someThing"></my-component>

<!-- 单次 prop 绑定 -->
<my-component :prop.once="someThing"></my-component>
```

- 另见：
  - Class 和 Style 绑定
  - 组件 Props

v-model

- 类型：随表单控件类型不同而不同。

- 限制：
    - `<input>`
    - `<select>`
    - `<textarea>`
- **Param Attributes:**
    - `lazy`
    - `number`
    - `debounce`
- 用法：

  在表单控件上创建双向绑定。

- 另见：表单控件绑定

v-ref

- 不需要表达式
- 限制：子组件
- 参数：`id (required)`
- 用法：

  在父组件上注册一个子组件的索引，便于直接访问。不需要表达式。必须提供参数 id。可以通过父组件的 `$refs` 对象访问子组件。

  在和 `v-for` 一起用时，注册的值将是一个数组，包含所有的子组件，对应于绑定数组。如果 `v-for` 用在一个对象上，注册的值将是一个对象，包含所有的子组件，对应于绑定对象。

- 注意：

  因为 HTML 不区分大小写，camelCase 名字比如 `v-ref:someRef` 将全转为小写。可以用 `v-ref:some-ref` 设置 `this.$els.someRef` 。

- 示例：

```
<comp v-ref:child></comp>
<comp v-ref:some-child></comp>
```

```
// 从父组件访问
this.$refs.child
this.$refs.someChild
```

  使用 `v-for` ：

```
<comp v-ref:list v-for="item in list"></comp>
```

```
// 值是一个数组
this.$refs.list
```

- 另见：子组件索引

v-el

- 不需要表达式
- 参数：`id`（必需）
- 用法：

  为 DOM 元素注册一个索引，方便通过所属实例的 `$els` 访问这个元素。

- 注意：

  因为 HTML 不区分大小写，camelCase 名字比如 `v-el:someEl` 将转为全小写。可以用 `v-el:some-el` 设置 `this.$els.someEl`。

- 示例：

```
<span v-el:msg>hello</span>
<span v-el:other-msg>world</span>
```

```
this.$els.msg.textContent // -> "hello"
this.$els.otherMsg.textContent // -> "world"
```

v-pre

- 不需要表达式
- 用法：

  跳过编译这个元素和它的子元素。可以用来显示原始 Mustache 标签。跳过大量没有指令的节点会加快编译。

- 示例：

```
<span v-pre>{{ this will not be compiled }}</span>
```

v-cloak

- 不需要表达式
- 用法：

  这个指令保持在元素上直到关联实例结束编译。和 CSS 规则如 `[v-cloak] { display: none }` 一起用时，这个指令可以隐藏未编译的 Mustache 标签直到实例准备完毕。

- 示例：

```
[v-cloak] {
 display: none;
}
```

```
<div v-cloak>
 {{ message }}
</div>
```

  `<div>` 不会显示，直到编译结束。

特殊元素

component

- 特性：
  - `is`
- 用法：

  另一种调用组件的语法。主要是和 `is` 特性一起用于动态组件。

```
<!-- 动态组件 -->
<!-- 由实例的 `componentId` 属性控制 -->
<component :is="componentId"></component>
```

- 另见：动态组件

slot

- 特性：
  - `name`

- 用法：

  `<slot>` 元素作为组件模板之中的内容分发插槽。这个元素自身将被替换。

  有 `name` 特性的 slot 称为命名 slot。 有 `slot` 特性的内容将分发到名字相匹配的命名 slot。

- 另见：使用 slot 分发内容

partial

- 特性：
  - `name`
- 用法：

  `<partial>` 元素是已注册的 partial 的插槽，partial 在插入时被 Vue 编译。 `<partial>` 元素本身会被替换。 `<partial>` 元素需要指定 `name` 特性。

- 示例：

```
// 注册 partial
Vue.partial('my-partial', '<p>This is a partial! {{msg}}</p>')
```

```
<!-- 静态 partial -->
<partial name="my-partial"></partial>

<!-- 动态 partial -->
<!-- 渲染 partial，id === vm.partialId -->
<partial v-bind:name="partialId"></partial>

<!-- 动态 partial，使用 v-bind 缩写语法 -->
<partial :name="partialId"></partial>
```

过滤器

capitalize

- 示例：

```
{{ msg &#124; capitalize }}
```

'abc' => 'Abc'

uppercase

- 示例：

  ```
  {{ msg &#124; uppercase }}
  ```

  'abc' => 'ABC'

lowercase

- 示例：

  ```
  {{ msg &#124; lowercase }}
  ```

  'ABC' => 'abc'

currency

- 参数：
  - `{String} [symbol] - default: '$'`
- 示例：

  ```
  {{ amount &#124; currency }}
  ```

  12345 => $12,345.00

  使用其它符号：

  ```
  {{ amount &#124; currency '£' }}
  ```

  12345 => £12,345.00

pluralize

- 参数：
  - `{String} single, [double, triple, ...]`
- 用法：

  如果只有一个参数，复数形式只是简单地在末尾添加一个 "s"。如果有多个参数，参数被当作一个字符串数组，对应一个、两个、三个...复数词。如果值的个数多于参数的个数，多出的使用最后一个参数。

- 示例：

  ```
  {{count}} {{count &#124; pluralize 'item'}}
  ```

  1 => ‘1 item’ 2 => ‘2 items’

  ```
  {{date}}{{date &#124; pluralize 'st' 'nd' 'rd' 'th'}}
  ```

  结果：

  1 => ‘1st’ 2 => ‘2nd’ 3 => ‘3rd’ 4 => ‘4th’ 5 => ‘5th’

json

- 参数：
  - `{Number} [indent] - 默认值：2`
- 用法：

  输出经 `JSON.stringify()` 处理后的结果，而不是输出 `toString()` 的结果（如 `[object Object]`）。

- 示例：

  以四个空格的缩进打印一个对象：

  ```
  <pre>{{ nestedObject &#124; json 4 }}</pre>
  ```

debounce

- 限制：指令的值须是函数，如 `v-on`
- 参数：
  - `{Number} [wait] - 默认值： 300`
- 用法：

  包装处理器，让它延迟执行 `x` ms， 默认延迟 300ms。包装后的处理器在调用之后至少将延迟 `x` ms， 如果在延迟结束前再次调用，延迟时长重置为 `x` ms。

- 示例：

  ```
  <input @keyup="onKeyup &#124; debounce 500">
  ```

limitBy

- 限制：指令的值须是数组，如 `v-for`
- 参数：
  - `{Number} limit`
  - `{Number} [offset]`
- 用法：

  限制数组为开始 N 个元素，N 由第一个参数指定。第二个参数是可选的，指定开始的偏移量。

  ```
  <!-- 只显示开始 10 个元素 -->
  <div v-for="item in items &#124; limitBy 10"></div>

  <!-- 显示第 5 到 15 元素-->
  <div v-for="item in items &#124; limitBy 10 5"></div>
  ```

filterBy

- 限制：指令的值须是数组，如 `v-for`
- 参数：
  - `{String | Function} targetStringOrFunction`
  - `"in" (optional delimiter)`
  - `{String} [...searchKeys]`
- 用法：

  返回过滤后的数组。第一个参数可以是字符串或函数。

  如果第一个参数是字符串，则在每个数组元素中搜索它：

  ```
  <div v-for="item in items &#124; filterBy 'hello'">
  ```

  在上例中，只显示包含字符串 `"hello"` 的元素。

  如果 item 是一个对象，过滤器将递归地在它所有属性中搜索。为了缩小搜索范围，可以指定一个搜索字段：

  ```
  <div v-for="user in users &#124; filterBy 'Jack' in 'name'">
  ```

  在上例中，过滤器只在用户对象的 `name` 属性中搜索 `"Jack"`。为了更好的性能，最好始终限制搜索范围。

上例使用静态参数，当然可以使用动态参数作为搜索目标或搜索字段。配合 `v-model` 我们可以轻松实现输入提示效果：

```
<div id="filter-by-example">
 <input v-model="name">
 <ul>
 <li v-for="user in users &#124; filterBy name in 'name'">
 {{ user.name }}
 </li>
 </ul>
</div>
```

```
new Vue({
 el: '#filter-by-example',
 data: {
 name: '',
 users: [
 { name: 'Bruce' },
 { name: 'Chuck' },
 { name: 'Jackie' }
 ]
 }
})
```

- Bruce
- Chuck
- Jackie

- 另一个示例：

  多搜索字段：

  ```
  <li v-for="user in users &#124; filterBy searchText in 'name' '
  ```

  多搜索字段为一个动态数组：

  ```
  <!-- fields = ['fieldA', 'fieldB'] -->
  <div v-for="user in users &#124; filterBy searchText in fields"
  ```

使用自定义过滤函数：

```
<div v-for="user in users &#124; filterBy myCustomFilterFunctio
```

orderBy

- 限制：指令的值须是数组，如 `v-for`
- 参数：
  - `{String} sortKey`
  - `{String} [order] - 默认值：1`
- 用法：

  返回排序后的数组。`sortKey` 是用于排序的字段。可选参数 `order` 决定结果升序（`order >= 0`）或降序（`order < 0`）。

  对于原始类型数组，`sortKey` 可以是任意的真值。

- 示例：

  按名字排序用户：

```
<ul>
 <li v-for="user in users &#124; orderBy 'name'">
 {{ user.name }}
 </li>
</ul>
```

  降序：

```
<ul>
 <li v-for="user in users &#124; orderBy 'name' -1">
 {{ user.name }}
 </li>
</ul>
```

  原始类型数组：

```
<ul>
 <li v-for="n in numbers &#124; orderBy true">
 {{ n }}
 </li>
</ul>
```

动态排序：

```
<div id="orderby-example">
 <button @click="order = order * -1">Reverse Sort Order</button
 <ul>
 <li v-for="user in users &#124; orderBy 'name' order">
 {{ user.name }}
 </li>
 </ul>
</div>
```

```
new Vue({
 el: '#orderby-example',
 data: {
 order: 1,
 users: [{ name: 'Bruce' }, { name: 'Chuck' }, { name: 'Jackie'
 }
})
```

Reverse Sort Order

- Bruce
- Chuck
- Jackie

示例

Markdown 编辑器 Example

> 极简的 Markdown 编辑器。

```
# hello
```

hello

html

```
<div id="editor">
  <textarea v-model="input" debounce="300"></textarea>
  <div v-html="input | marked"></div>
</div>
```

js

```
new Vue({
  el: '#editor',
  data: {
    input: '# hello'
  },
  filters: {
    marked: marked
  }
})
```

css

```
html, body, #editor {
  margin: 0;
  height: 100%;
  font-family: 'Helvetica Neue', Arial, sans-serif;
  color: #333;
}

textarea, #editor div {
  display: inline-block;
  width: 49%;
  height: 100%;
  vertical-align: top;
  -webkit-box-sizing: border-box;
  -moz-box-sizing: border-box;
  box-sizing: border-box;
  padding: 0 20px;
}

textarea {
  border: none;
  border-right: 1px solid #ccc;
  resize: none;
  outline: none;
  background-color: #f6f6f6;
  font-size: 14px;
  font-family: 'Monaco', courier, monospace;
  padding: 20px;
}

code {
  color: #f66;
}
```

GitHub 提交 Example

> 这个例子通过 GitHub 的 API 获取 Vue.js 最近的提交记录，并展示为一个列表。你可以在 master 和 dev 分支之间切换。

Latest Vue.js Commits

◉ master ○ dev

vuejs/vue@master

- da9e036 - [release] 1.0.13
  by **Evan You** at **2015-12-24 22:39:37**
- 8c77fb8 - [build] 1.0.13
  by **Evan You** at **2015-12-24 22:39:36**
- 03b580e - use pre-hook for setting up prop two-way sync (fix #2064)
  by **Evan You** at **2015-12-24 22:28:14**

html

```
<div id="demo">
  <h1>Latest Vue.js Commits</h1>
  <template v-for="branch in branches">
    <input type="radio"
      name="branch"
      :id="branch"
      :value="branch"
      v-model="currentBranch">
    <label :for="branch">{{branch}}</label>
  </template>
  <p>vuejs/vue@{{currentBranch}}</p>
  <ul>
    <li v-for="record in commits">
      <a :href="record.html_url" target="_blank" class="commit">{{r
      - <span class="message">{{record.commit.message | truncate}}<
      by <span class="author">{{record.commit.author.name}}</span>
      at <span class="date">{{record.commit.author.date | formatDat
    </li>
  </ul>
</div>
```

js

```
var apiURL = 'https://api.github.com/repos/vuejs/vue/commits?per_pa

var demo = new Vue({

  el: '#demo',

  data: {
    branches: ['master', 'dev'],
    currentBranch: 'master',
    commits: null
  },

  created: function () {
    this.fetchData()
  },

  watch: {
    currentBranch: 'fetchData'
  },

  filters: {
    truncate: function (v) {
      var newline = v.indexOf('\n')
      return newline > 0 ? v.slice(0, newline) : v
    },
    formatDate: function (v) {
      return v.replace(/T|Z/g, ' ')
    }
  },

  methods: {
    fetchData: function () {
      var xhr = new XMLHttpRequest()
      var self = this
      xhr.open('GET', apiURL + self.currentBranch)
      xhr.onload = function () {
        self.commits = JSON.parse(xhr.responseText)
      }
      xhr.send()
    }
  }
})
```

CSS

```
#demo {
  font-family: 'Helvetica', Arial, sans-serif;
}
a {
  text-decoration: none;
  color: #f66;
}
li {
  line-height: 1.5em;
  margin-bottom: 20px;
}
.author, .date {
  font-weight: bold;
}
```

Firebase + 验证 Example

> 这个例子使用 Firebase 作为后台数据存储，并与客户端实时同步 (你可以在多个浏览器标签中打开它)。另外，它使用计算属性实时校验，并且在添加/删除项目时触发 CSS 过渡效果。

Add User

Name cannot be empty.
Please provide a valid email address.

html

```
<div id="app">
  <ul>
    <li class="user" v-for="user in users" transition>
      <span>{{user.name}} - {{user.email}}</span>
      <button v-on:click="removeUser(user)">X</button>
    </li>
  </ul>
  <form id="form" v-on:submit.prevent="addUser">
    <input v-model="newUser.name">
    <input v-model="newUser.email">
    <input type="submit" value="Add User">
  </form>
  <ul class="errors">
    <li v-show="!validation.name">Name cannot be empty.</li>
    <li v-show="!validation.email">Please provide a valid email add
  </ul>
</div>
```

js

```
var baseURL = 'https://vue-demo.firebaseIO.com/'
var emailRE = /^(([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)

/**
 * Setup firebase sync
 */

var Users = new Firebase(baseURL + 'users')

Users.on('child_added', function (snapshot) {
  var item = snapshot.val()
  item.id = snapshot.key()
  app.users.push(item)
})

Users.on('child_removed', function (snapshot) {
  var id = snapshot.key()
  app.users.some(function (user) {
    if (user.id === id) {
      app.users.$remove(user)
      return true
    }
  })
})

/**
 * Create Vue app
 */

var app = new Vue({
```

```
  // element to mount to
  el: '#app',

  // initial data
  data: {
    users: [],
    newUser: {
      name: '',
      email: ''
    }
  },

  // computed property for form validation state
  computed: {
    validation: function () {
      return {
        name: !!this.newUser.name.trim(),
        email: emailRE.test(this.newUser.email)
      }
    },
    isValid: function () {
      var validation = this.validation
      return Object.keys(validation).every(function (key) {
        return validation[key]
      })
    }
  },

  // methods
  methods: {
    addUser: function () {
      if (this.isValid) {
        Users.push(this.newUser)
        this.newUser.name = ''
        this.newUser.email = ''
      }
    },
    removeUser: function (user) {
      new Firebase(baseURL + 'users/' + user.id).remove()
    }
  }
})
```

CSS

```
body {
  font-family: Helvetica, Arial, sans-serif;
}

ul {
  padding: 0;
}

.user {
  height: 30px;
  line-height: 30px;
  padding: 10px;
  border-top: 1px solid #eee;
  overflow: hidden;
  transition: all .25s ease;
}

.user:last-child {
  border-bottom: 1px solid #eee;
}

.v-enter, .v-leave {
  height: 0;
  padding-top: 0;
  padding-bottom: 0;
  border-top-width: 0;
  border-bottom-width: 0;
}

.errors {
  color: #f00;
}
```

表格组件 Example

> 这个例子创建了一个可复用的表格组件，用外部数据使用它。

Search

Name ▲	Power ▲
Chuck Norris	Infinity
Bruce Lee	9000
Jackie Chan	7000
Jet Li	8000

html

```
<!-- component template -->
<script type="text/x-template" id="grid-template">
  <table>
    <thead>
      <tr>
        <th v-for="key in columns"
          @click="sortBy(key)"
          :class="{active: sortKey == key}">
          {{key | capitalize}}
          <span class="arrow"
            :class="sortOrders[key] > 0 ? 'asc' : 'dsc'">
          </span>
        </th>
      </tr>
    </thead>
    <tbody>
      <tr v-for="
        entry in data
        | filterBy filterKey
        | orderBy sortKey sortOrders[sortKey]">
        <td v-for="key in columns">
          {{entry[key]}}
        </td>
      </tr>
    </tbody>
  </table>
</script>

<!-- demo root element -->
<div id="demo">
  <form id="search">
    Search <input name="query" v-model="searchQuery">
  </form>
  <demo-grid
    :data="gridData"
    :columns="gridColumns"
    :filter-key="searchQuery">
  </demo-grid>
</div>
```

js

```
// register the grid component
Vue.component('demo-grid', {
  template: '#grid-template',
  props: {
    data: Array,
    columns: Array,
    filterKey: String
  },
  data: function () {
    var sortOrders = {}
    this.columns.forEach(function (key) {
      sortOrders[key] = 1
    })
    return {
      sortKey: '',
      sortOrders: sortOrders
    }
  },
  methods: {
    sortBy: function (key) {
      this.sortKey = key
      this.sortOrders[key] = this.sortOrders[key] * -1
    }
  }
})

// bootstrap the demo
var demo = new Vue({
  el: '#demo',
  data: {
    searchQuery: '',
    gridColumns: ['name', 'power'],
    gridData: [
      { name: 'Chuck Norris', power: Infinity },
      { name: 'Bruce Lee', power: 9000 },
      { name: 'Jackie Chan', power: 7000 },
      { name: 'Jet Li', power: 8000 }
    ]
  }
})
```

css

```
body {
  font-family: Helvetica Neue, Arial, sans-serif;
  font-size: 14px;
  color: #444;
}

table {
  border: 2px solid #42b983;
```

```
  border-radius: 3px;
  background-color: #fff;
}

th {
  background-color: #42b983;
  color: rgba(255,255,255,0.66);
  cursor: pointer;
  -webkit-user-select: none;
  -moz-user-select: none;
  -user-select: none;
}

td {
  background-color: #f9f9f9;
}

th, td {
  min-width: 120px;
  padding: 10px 20px;
}

th.active {
  color: #fff;
}

th.active .arrow {
  opacity: 1;
}

.arrow {
  display: inline-block;
  vertical-align: middle;
  width: 0;
  height: 0;
  margin-left: 5px;
  opacity: 0.66;
}

.arrow.asc {
  border-left: 4px solid transparent;
  border-right: 4px solid transparent;
  border-bottom: 4px solid #fff;
}

.arrow.dsc {
  border-left: 4px solid transparent;
  border-right: 4px solid transparent;
  border-top: 4px solid #fff;
}

#search {
  margin-bottom: 10px;
```

```
}
```

树状视图 Example

> 这个例子实现了一个简单的树状视图，演示如何递归使用组件。

(You can double click on an item to turn it into a folder.)

- **My Tree [-]**
  - hello
  - wat
  - **child folder [-]**
    - **child folder [-]**
      - hello
      - wat
      - +
    - hello
    - wat
    - **child folder [-]**
      - hello
      - wat
      - +
    - +
  - +

html

```
<!-- item template -->
<script type="text/x-template" id="item-template">
  <li>
    <div
      :class="{bold: isFolder}"
      @click="toggle"
      @dblclick="changeType">
      {{model.name}}
      <span v-if="isFolder">[{{open ? '-' : '+'}}]</span>
    </div>
    <ul v-show="open" v-if="isFolder">
      <item
        class="item"
        v-for="model in model.children"
        :model="model">
      </item>
      <li @click="addChild">+</li>
    </ul>
  </li>
</script>

<p>(You can double click on an item to turn it into a folder.)</p>

<!-- the demo root element -->
<ul id="demo">
  <item
    class="item"
    :model="treeData">
  </item>
</ul>
```

js

```
// demo data
var data = {
  name: 'My Tree',
  children: [
    { name: 'hello' },
    { name: 'wat' },
    {
      name: 'child folder',
      children: [
        {
          name: 'child folder',
          children: [
            { name: 'hello' },
            { name: 'wat' }
          ]
        },
        { name: 'hello' },
```

```
        { name: 'wat' },
        {
          name: 'child folder',
          children: [
            { name: 'hello' },
            { name: 'wat' }
          ]
        }
      ]
    }
  ]
}

// define the item component
Vue.component('item', {
  template: '#item-template',
  props: {
    model: Object
  },
  data: function () {
    return {
      open: false
    }
  },
  computed: {
    isFolder: function () {
      return this.model.children &&
        this.model.children.length
    }
  },
  methods: {
    toggle: function () {
      if (this.isFolder) {
        this.open = !this.open
      }
    },
    changeType: function () {
      if (!this.isFolder) {
        Vue.set(this.model, 'children', [])
        this.addChild()
        this.open = true
      }
    },
    addChild: function () {
      this.model.children.push({
        name: 'new stuff'
      })
    }
  }
})

// boot up the demo
var demo = new Vue({
```

```
  el: '#demo',
  data: {
    treeData: data
  }
})
```

css

```
body {
  font-family: Menlo, Consolas, monospace;
  color: #444;
}
.item {
  cursor: pointer;
}
.bold {
  font-weight: bold;
}
ul {
  padding-left: 1em;
  line-height: 1.5em;
  list-style-type: dot;
}
```

SVG 图形 Example

> 这个例子演示了综合使用自定义组件、计算属性、双向绑定及 SVG 支持。

A
F
B
E
C
D

A 100 X
B 100 X
C 100 X
D 100 X
E 100 X
F 100 X

Add a Stat

* input[type="range"] requires IE10 or above.


```
[
  {
    "label": "A",
    "value": 100
  },
  {
    "label": "B",
    "value": 100
  },
  {
    "label": "C",
    "value": 100
  },
  {
    "label": "D",
    "value": 100
  },
  {
    "label": "E",
    "value": 100
  },
  {
    "label": "F",
    "value": 100
  }
]
```


html

```
<!-- template for the polygraph component. -->
<script type="text/x-template" id="polygraph-template">
  <g>
    <polygon :points="points"></polygon>
    <circle cx="100" cy="100" r="80"></circle>
    <axis-label
      v-for="stat in stats"
      :stat="stat"
      :index="$index"
      :total="stats.length">
    </axis-label>
  </g>
</script>

<!-- template for the axis label component. -->
<script type="text/x-template" id="axis-label-template">
  <text :x="point.x" :y="point.y">{{stat.label}}</text>
</script>

<!-- demo root element -->
<div id="demo">
  <!-- Use the component -->
  <svg width="200" height="200">
    <polygraph :stats="stats"></polygraph>
  </svg>
  <!-- controls -->
  <div v-for="stat in stats">
    <label>{{stat.label}}</label>
    <input type="range" v-model="stat.value" min="0" max="100">
    <span>{{stat.value}}</span>
    <button @click="remove(stat)">X</button>
  </div>
  <form id="add">
    <input name="newlabel" v-model="newLabel">
    <button @click="add">Add a Stat</button>
  </form>
  <pre id="raw">{{stats | json}}</pre>
</div>

<p style="font-size:12px">* input[type="range"] requires IE10 or ab
```

js

```
// The raw data to observe
var stats = [
  { label: 'A', value: 100 },
  { label: 'B', value: 100 },
  { label: 'C', value: 100 },
  { label: 'D', value: 100 },
  { label: 'E', value: 100 },
```

```
  { label: 'F', value: 100 }
]

// A resusable polygon graph component
Vue.component('polygraph', {
  props: ['stats'],
  template: '#polygraph-template',
  replace: true,
  computed: {
    // a computed property for the polygon's points
    points: function () {
      var total = this.stats.length
      return this.stats.map(function (stat, i) {
        var point = valueToPoint(stat.value, i, total)
        return point.x + ',' + point.y
      }).join(' ')
    }
  },
  components: {
    // a sub component for the labels
    'axis-label': {
      props: {
        stat: Object,
        index: Number,
        total: Number
      },
      template: '#axis-label-template',
      replace: true,
      computed: {
        point: function () {
          return valueToPoint(
            +this.stat.value + 10,
            this.index,
            this.total
          )
        }
      }
    }
  }
})

// math helper...
function valueToPoint (value, index, total) {
  var x     = 0
  var y     = -value * 0.8
  var angle = Math.PI * 2 / total * index
  var cos   = Math.cos(angle)
  var sin   = Math.sin(angle)
  var tx    = x * cos - y * sin + 100
  var ty    = x * sin + y * cos + 100
  return {
    x: tx,
    y: ty
```

```
    }
  }

  // bootstrap the demo
  new Vue({
    el: '#demo',
    data: {
      newLabel: '',
      stats: stats
    },
    methods: {
      add: function (e) {
        e.preventDefault()
        if (!this.newLabel) return
        this.stats.push({
          label: this.newLabel,
          value: 100
        })
        this.newLabel = ''
      },
      remove: function (stat) {
        if (this.stats.length > 3) {
          this.stats.$remove(stat)
        } else {
          alert('Can\'t delete more!')
        }
      }
    }
  })
```

css

```
body {
    font-family: Helvetica Neue, Arial, sans-serif;
}

polygon {
    fill: #42b983;
    opacity: .75;
}

circle {
    fill: transparent;
    stroke: #999;
}

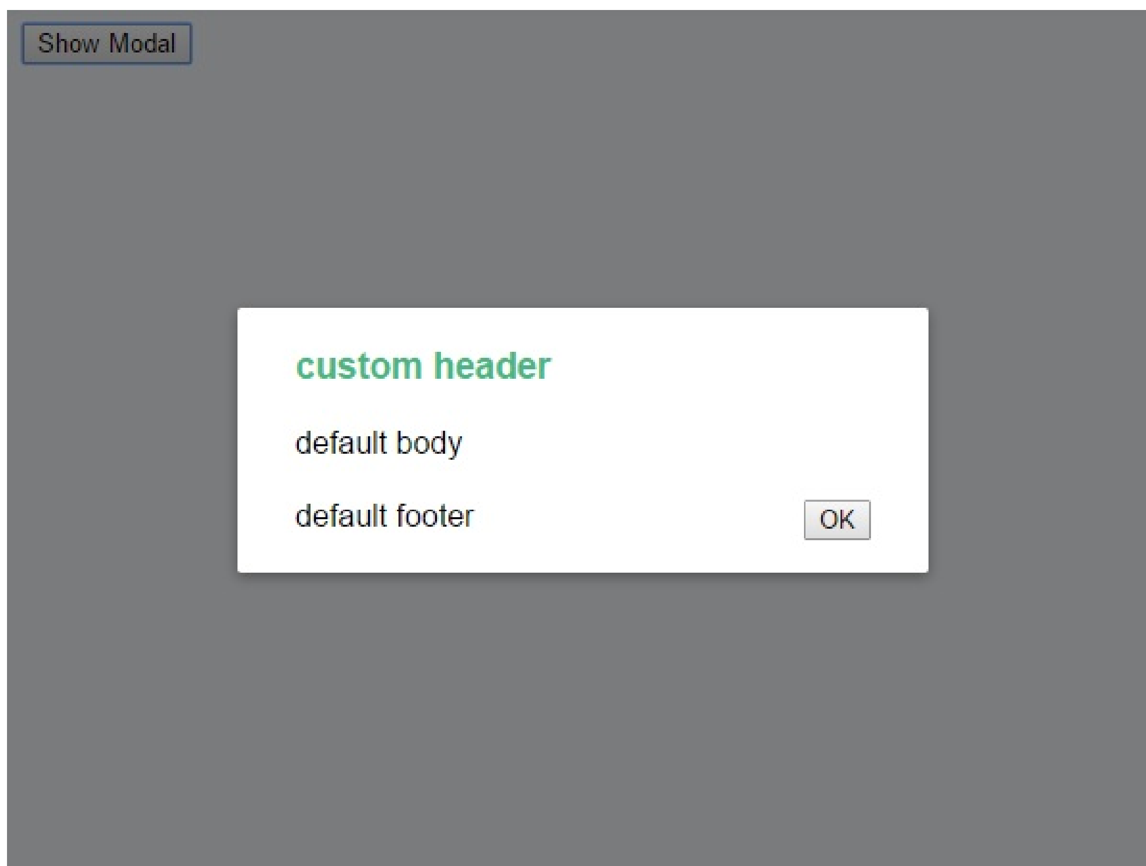
text {
    font-family: Helvetica Neue, Arial, sans-serif;
    font-size: 10px;
    fill: #666;
}

label {
    display: inline-block;
    margin-left: 10px;
    width: 20px;
}

#raw {
    position: absolute;
    top: 0;
    left: 300px;
}
```

模态组件 Example

> 用到的特性：组件、prop 传递、内容传入、过渡。



html

```
<!-- template for the modal component -->
<script type="x/template" id="modal-template">
  <div class="modal-mask" v-show="show" transition="modal">
    <div class="modal-wrapper">
      <div class="modal-container">

        <div class="modal-header">
          <slot name="header">
            default header
          </slot>
        </div>

        <div class="modal-body">
          <slot name="body">
            default body
          </slot>
        </div>

        <div class="modal-footer">
          <slot name="footer">
            default footer
            <button class="modal-default-button"
              @click="show = false">
              OK
            </button>
          </slot>
        </div>
      </div>
    </div>
  </div>
</script>

<!-- app -->
<div id="app">
  <button id="show-modal" @click="showModal = true">Show Modal</but
  <!-- use the modal component, pass in the prop -->
  <modal :show.sync="showModal">
    <!--
      you can use custom content here to overwrite
      default content
    -->
    <h3 slot="header">custom header</h3>
  </modal>
</div>
```

js

```
// register modal component
Vue.component('modal', {
  template: '#modal-template',
  props: {
    show: {
      type: Boolean,
      required: true,
      twoWay: true
    }
  }
})

// start app
new Vue({
  el: '#app',
  data: {
    showModal: false
  }
})
```

css

```
.modal-mask {
  position: fixed;
  z-index: 9998;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  background-color: rgba(0, 0, 0, .5);
  display: table;
  transition: opacity .3s ease;
}

.modal-wrapper {
  display: table-cell;
  vertical-align: middle;
}

.modal-container {
  width: 300px;
  margin: 0px auto;
  padding: 20px 30px;
  background-color: #fff;
  border-radius: 2px;
  box-shadow: 0 2px 8px rgba(0, 0, 0, .33);
  transition: all .3s ease;
  font-family: Helvetica, Arial, sans-serif;
}

.modal-header h3 {
```

```
  margin-top: 0;
  color: #42b983;
}

.modal-body {
  margin: 20px 0;
}

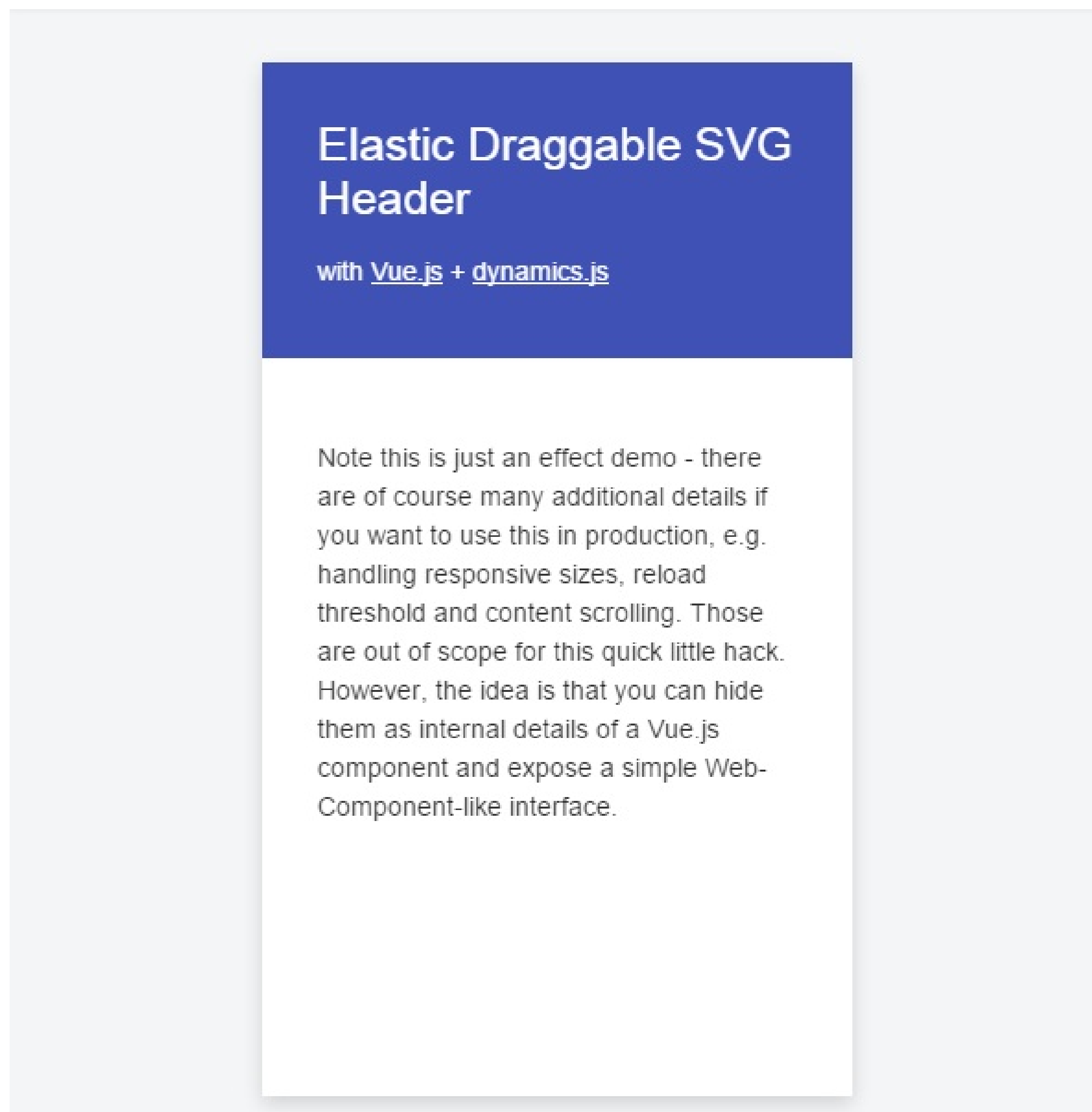
.modal-default-button {
  float: right;
}

/*
 * the following styles are auto-applied to elements with
 * v-transition="modal" when their visiblity is toggled
 * by Vue.js.
 *
 * You can easily play with the modal transition by editing
 * these styles.
 */

.modal-enter, .modal-leave {
  opacity: 0;
}

.modal-enter .modal-container,
.modal-leave .modal-container {
  -webkit-transform: scale(1.1);
  transform: scale(1.1);
}
```

Elastic Header Example



html

```
<!-- template for the component -->
<script type="x/template" id="header-view-template">
  <div class="draggable-header-view"
    @mousedown="startDrag" @touchstart="startDrag"
    @mousemove="onDrag" @touchmove="onDrag"
    @mouseup="stopDrag" @touchend="stopDrag" @mouseleave="stopDrag'
    <svg class="bg" width="320" height="560">
      <path :d="headerPath" fill="#3F51B5"></path>
    </svg>
    <div class="header">
      <slot name="header"></slot>
    </div>
    <div class="content" :style="contentPosition">
      <slot name="content"></slot>
    </div>
  </div>
</script>

<draggable-header-view>
  <template slot="header">
    <h1>Elastic Draggable SVG Header</h1>
    <p>with <a href="http://vuejs.org" target="_blank">Vue.js</a> -
  </template>
  <template slot="content">
    <p>Note this is just an effect demo - there are of course many
  </template>
</draggable-header-view>
```

js

```
document.body.addEventListener('touchmove', function (e) {
  e.preventDefault()
})

Vue.component('draggable-header-view', {
  template: '#header-view-template',
  data: function () {
    return {
      dragging: false,
      // quadratic bezier control point
      c: { x: 160, y: 160 },
      // record drag start point
      start: { x: 0, y: 0 }
    }
  },
  computed: {
    headerPath: function () {
      return 'M0,0 L320,0 320,160' +
        'Q' + this.c.x + ',' + this.c.y +
        ' 0,160'
```

```
    },
    contentPosition: function () {
      var dy = this.c.y - 160
      var dampen = dy > 0 ? 2 : 4
      return {
        transform: 'translate3d(0,' + dy / dampen + 'px,0)'
      }
    }
  },
  methods: {
    startDrag: function (e) {
      e = e.changedTouches ? e.changedTouches[0] : e
      this.dragging = true
      this.start.x = e.pageX
      this.start.y = e.pageY
    },
    onDrag: function (e) {
      e = e.changedTouches ? e.changedTouches[0] : e
      if (this.dragging) {
        this.c.x = 160 + (e.pageX - this.start.x)
        // dampen vertical drag by a factor
        var dy = e.pageY - this.start.y
        var dampen = dy > 0 ? 1.5 : 4
        this.c.y = 160 + dy / dampen
      }
    },
    stopDrag: function () {
      if (this.dragging) {
        this.dragging = false
        dynamics.animate(this.c, {
          x: 160,
          y: 160
        }, {
          type: dynamics.spring,
          duration: 700,
          friction: 280
        })
      }
    }
  }
})

new Vue({ el: 'body' })
```

CSS

```
h1 {
  font-weight: 300;
  font-size: 1.8em;
  margin-top: 0;
}
a {
  color: #fff;
}
.draggable-header-view {
  background-color: #fff;
  box-shadow: 0 4px 16px rgba(0,0,0,.15);
  width: 320px;
  height: 560px;
  overflow: hidden;
  margin: 30px auto;
  position: relative;
  font-family: 'Roboto', Helvetica, Arial, sans-serif;
  color: #fff;
  font-size: 14px;
  font-weight: 300;
  -webkit-user-select: none;
  -moz-user-select: none;
  -ms-user-select: none;
  user-select: none;
}
.draggable-header-view .bg {
  position: absolute;
  top: 0;
  left: 0;
  z-index: 0;
}
.draggable-header-view .header, .draggable-header-view .content {
  position: relative;
  z-index: 1;
  padding: 30px;
  box-sizing: border-box;
}
.draggable-header-view .header {
  height: 160px;
}
.draggable-header-view .content {
  color: #333;
  line-height: 1.5em;
}
```

自定义指令 Example

> 在这个例子中，Vue.js 与一个第三方 jQuery 插件(select2) 整合，通过将它包装到一个自定义指令中实现。

Selected: 0

default

html

```
<div id="el">
  <p>Selected: {{selected}}</p>
  <select v-select="selected" :options="options">
    <option value="0">default</option>
  </select>
</div>
```

js

```
Vue.directive('select', {
  twoWay: true,
  priority: 1000,

  params: ['options'],

  bind: function () {
    var self = this
    $(this.el)
      .select2({
        data: this.params.options
      })
      .on('change', function () {
        self.set(this.value)
      })
  },
  update: function (value) {
    $(this.el).val(value).trigger('change')
  },
  unbind: function () {
    $(this.el).off().select2('destroy')
  }
})

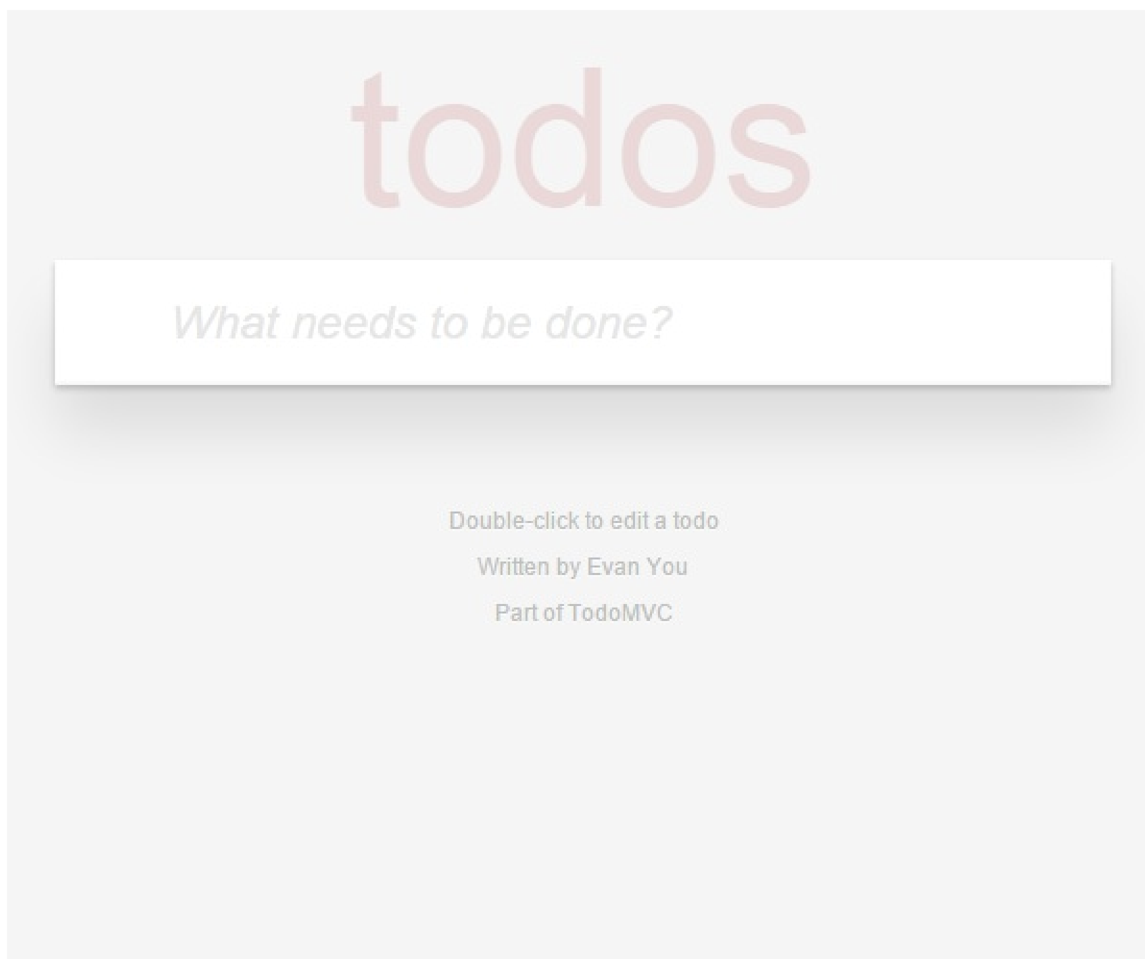
var vm = new Vue({
  el: '#el',
  data: {
    selected: 0,
    options: [
      { id: 1, text: 'hello' },
      { id: 2, text: 'what' }
    ]
  }
})
```

CSS

```
select {
  min-width: 300px;
}
```

TodoMVC Example

> 这是一个完全符合规范的 TodoMVC 实现，JavaScript 代码不超过 120 行 (除去注释和空行)。



> [Source]

HackerNews 克隆 Example

> 这是 HackerNews 的克隆，建立在 HN 官方的 Firebase API 上，并且使用 Webpack + vue-loader 构建。

Y **Hacker News** Built with Vue.js | Source

1. Sometimes, it’s just time to go home (www.benmilne.com)
   62 points by johns 2 hours ago | comments
2. Welcome, Ali (blog.ycombinator.com)
   123 points by sama 3 hours ago | comments
3. How to Start a Startup – Lecture 14: How to Operate (startupclass.samaltman.com)
   64 points by kqr2 4 hours ago | comments
4. Amazon Echo (www.amazon.com)
   747 points by danielsamuels 10 hours ago | comments
5. Confluent, a company for Apache Kafka and realtime data (www.linkedin.com)
   18 points by sbilstein 6 hours ago | comments
6. Running a South Pole data center (arstechnica.com)
   124 points by davidw 16 hours ago | comments
7. Colorado communities secure the right to build their own broadband (www.washingtonpost.com)
   253 points by Libertatea 11 hours ago | comments
8. Building a simple VGA-adapter for an 8-bit self made computer (tldr.fi)
   88 points by Arjuna 8 hours ago | comments
9. Show HN: StackShare – discover and discuss software stacks (stackshare.io)
   149 points by yonasb 10 hours ago | comments
10. On Pins and Needles: Stylist Turns Ancient Hairdo Debate on Its Head (2013) (online.wsj.com)
    81 points by gwern 9 hours ago | comments
11. Microsoft Changes Tack, Making Office Suite Free on Mobile (www.nytimes.com)
    216 points by shitehawk 13 hours ago | comments

> [Source]